GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	./				
File:	napi-inl.h	Lines:	1067	1110	96.1 %
Date:	2018-09-26 16:45:42	Branches:	847	1955	43.3 %

Line	Branch	Exec	Source
1			#ifndef SRC_NAPI_INL_H_
2			#define SRC_NAPI_INL_H_
3			
4			//
5			// N-API C++ Wrapper Classes
6			//
7			// Inline header-only implementations for "N-API" ABI-stable C APIs for Node.js.
8			//
9			
10			// Note: Do not include this file directly! Include "napi.h" instead.
11			
12			#include <cstring>
13			#include <type_traits>
14			
15			namespace Napi {
16			
17			// Helpers to handle functions exposed from C++.
18			namespace details {
19			
20			#ifdef NAPI_CPP_EXCEPTIONS
21			
22			#define NAPI_THROW(e) throw e
23			
24			// When C++ exceptions are enabled, Errors are thrown directly. There is no need
25			// to return anything after the throw statement. The variadic parameter is an
26			// optional return value that is ignored.
27			#define NAPI_THROW_IF_FAILED(env, status, ...) \
28			if ((status) != napi_ok) throw Error::New(env);
29			
30			#else // NAPI_CPP_EXCEPTIONS
31			
32			#define NAPI_THROW(e) (e).ThrowAsJavaScriptException();
33			
34			// When C++ exceptions are disabled, Errors are thrown as JavaScript exceptions,
35			// which are pending until the callback returns to JS. The variadic parameter
36			// is an optional return value; usually it is an empty result.
37			#define NAPI_THROW_IF_FAILED(env, status, ...) \
38			if ((status) != napi_ok) { \
39			Error::New(env).ThrowAsJavaScriptException(); \
40			return __VA_ARGS__; \
41			}
42			
43			#endif // NAPI_CPP_EXCEPTIONS
44			
45			#define NAPI_FATAL_IF_FAILED(status, location, message) \
46			do { \
47			if ((status) != napi_ok) { \
48			Error::Fatal((location), (message)); \
49			} \
50			} while (0)
51			
52			// For use in JS to C++ callback wrappers to catch any Napi::Error exceptions
53			// and rethrow them as JavaScript exceptions before returning from the callback.
54			template <typename Callable>
55		1924	inline napi_value WrapCallback(Callable callback) {
56			#ifdef NAPI_CPP_EXCEPTIONS
57			try {
	✓✓✗✓		
	✗✗✓✓		
	✗✓✗✗		
	✓✗✗✓		
	✗✗✓✗		
58	✗✓✗✓	1924	return callback();
	✗✗✓✗		
	✗✗✗✓		
	✗✗✗✓		
	✗✓✗✗		
	✗✓✗✓		
	✗✗✗		
59		106	} catch (const Error& e) {
60	✓✗✓✗	53	e.ThrowAsJavaScriptException();

```
✗✗✗✗
✗✗✗✗
   ✗✗
61              53    return nullptr;
62                  }
63                #else // NAPI_CPP_EXCEPTIONS
64                  // When C++ exceptions are disabled, errors are immediately thrown as JS
65                  // exceptions, so there is no need to catch and rethrow them here.
66                  return callback();
67                #endif // NAPI_CPP_EXCEPTIONS
68                }
69
70                template <typename Callable, typename Return>
71                struct CallbackData {
72                  static inline
73             1752  napi_value Wrapper(napi_env env, napi_callback_info info) {
74             1752    return details::WrapCallback([&] {
75 ✓✗✗✗        1752      CallbackInfo callbackInfo(env, info);
76                        CallbackData* callbackData =
77 ✓✗✗✗        1752        static_cast<CallbackData*>(callbackInfo.Data());
78 ✓✗✗✗        1752      callbackInfo.SetData(callbackData->data);
79 ✓✓✗✗        3471      return callbackData->callback(callbackInfo);
80             3504    });
81                  }
82
83                  Callable callback;
84                  void* data;
85                };
86
87                template <typename Callable>
88                struct CallbackData<Callable, void> {
89                  static inline
90              95  napi_value Wrapper(napi_env env, napi_callback_info info) {
91              95    return details::WrapCallback([&] {
92    ✓✗        95      CallbackInfo callbackInfo(env, info);
93                        CallbackData* callbackData =
94    ✓✗        95        static_cast<CallbackData*>(callbackInfo.Data());
95    ✓✗        95      callbackInfo.SetData(callbackData->data);
96    ✓✓        95      callbackData->callback(callbackInfo);
97              150     return nullptr;
98              190   });
99                  }
100
101                 Callable callback;
102                 void* data;
103               };
104
105               template <typename T, typename Finalizer, typename Hint = void>
106               struct FinalizeData {
107                 static inline
108             3   void Wrapper(napi_env env, void* data, void* finalizeHint) {
109             3     FinalizeData* finalizeData = static_cast<FinalizeData*>(finalizeHint);
110             3     finalizeData->callback(Env(env), static_cast<T*>(data));
111             3     delete finalizeData;
112             3   }
113
114                 static inline
115             3   void WrapperWithHint(napi_env env, void* data, void* finalizeHint) {
116             3     FinalizeData* finalizeData = static_cast<FinalizeData*>(finalizeHint);
117             3     finalizeData->callback(Env(env), static_cast<T*>(data), finalizeData->hint);
118             3     delete finalizeData;
119             3   }
120
121                 Finalizer callback;
122                 Hint* hint;
123               };
124
125               template <typename Getter, typename Setter>
126               struct AccessorCallbackData {
127                 static inline
128             12  napi_value GetterWrapper(napi_env env, napi_callback_info info) {
129             12    return details::WrapCallback([&] {
130    ✓✗       12      CallbackInfo callbackInfo(env, info);
131                       AccessorCallbackData* callbackData =
132    ✓✗       12        static_cast<AccessorCallbackData*>(callbackInfo.Data());
133    ✓✗       24      return callbackData->getterCallback(callbackInfo);
134             24    });
```

```
                }

                static inline
            6   napi_value SetterWrapper(napi_env env, napi_callback_info info) {
            6     return details::WrapCallback([&] {
✓✗          6       CallbackInfo callbackInfo(env, info);
                    AccessorCallbackData* callbackData =
✓✗          6         static_cast<AccessorCallbackData*>(callbackInfo.Data());
✓✗          6       callbackData->setterCallback(callbackInfo);
           12       return nullptr;
           12     });
                }

                Getter getterCallback;
                Setter setterCallback;
              };

              }  // namespace details

              ////////////////////////////////////////////////////////////////////////////////
              // Module registration
              ////////////////////////////////////////////////////////////////////////////////

              #define NODE_API_MODULE(modname, regfunc)                \
                napi_value __napi_ ## regfunc(napi_env env,            \
                                              napi_value exports) {    \
                  return Napi::RegisterModule(env, exports, regfunc);  \
                }                                                      \
                NAPI_MODULE(modname, __napi_ ## regfunc);

              // Adapt the NAPI_MODULE registration function:
              //  - Wrap the arguments in NAPI wrappers.
              //  - Catch any NAPI errors and rethrow as JS exceptions.
            1 inline napi_value RegisterModule(napi_env env,
                                               napi_value exports,
                                               ModuleRegisterCallback registerCallback) {
            1   return details::WrapCallback([&] {
                  return napi_value(registerCallback(Napi::Env(env),
✓✗✓✗        2                                    Napi::Object(env, exports)));
✓✗          3   });
              }

              ////////////////////////////////////////////////////////////////////////////////
              // Env class
              ////////////////////////////////////////////////////////////////////////////////

      2002218 inline Env::Env(napi_env env) : _env(env) {
      2002218 }

      2002572 inline Env::operator napi_env() const {
      2002572   return _env;
              }

            1 inline Object Env::Global() const {
                napi_value value;
✓✗✓✗        1   napi_status status = napi_get_global(*this, &value);
✗✓✗✗        1   NAPI_THROW_IF_FAILED(*this, status, Object());
  ✗✗
✓✗✓✗        1   return Object(*this, value);
              }

           53 inline Value Env::Undefined() const {
                napi_value value;
✓✗✓✗       53   napi_status status = napi_get_undefined(*this, &value);
✗✓✗✗       53   NAPI_THROW_IF_FAILED(*this, status, Value());
  ✗✗
✓✗✓✗       53   return Value(*this, value);
              }

              inline Value Env::Null() const {
                napi_value value;
                napi_status status = napi_get_null(*this, &value);
                NAPI_THROW_IF_FAILED(*this, status, Value());
                return Value(*this, value);
              }

            2 inline bool Env::IsExceptionPending() const {
                bool result;
```

```
211    ✓✗       2  napi_status status = napi_is_exception_pending(_env, &result);
212    ✗✓       2  if (status != napi_ok) result = false; // Checking for a pending exception shouldn't throw.
213              2  return result;
214               }
215
216               inline Error Env::GetAndClearPendingException() {
217                 napi_value value;
218                 napi_status status = napi_get_and_clear_last_exception(_env, &value);
219                 if (status != napi_ok) {
220                   // Don't throw another exception when failing to get the exception!
221                   return Error();
222                 }
223                 return Error(_env, value);
224               }
225
226               ////////////////////////////////////////////////////////////////////////////////
227               // Value class
228               ////////////////////////////////////////////////////////////////////////////////
229
230              5inline Value::Value() : _env(nullptr), _value(nullptr) {
231              5}
232
233        1007459inline Value::Value(napi_env env, napi_value value) : _env(env), _value(value) {
234        1007459}
235
236           3075inline Value::operator napi_value() const {
237           3075  return _value;
238               }
239
240            104inline bool Value::operator ==(const Value& other) const {
241            104  return StrictEquals(other);
242               }
243
244               inline bool Value::operator !=(const Value& other) const {
245                 return !this->operator ==(other);
246               }
247
248            104inline bool Value::StrictEquals(const Value& other) const {
249                 bool result;
250 ✓✗✓✗✓✗  104  napi_status status = napi_strict_equals(_env, *this, other, &result);
251 ✗✓✗✗    104  NAPI_THROW_IF_FAILED(_env, status, false);
252            104  return result;
253               }
254
255            305inline Napi::Env Value::Env() const {
256            305  return Napi::Env(_env);
257               }
258
259              2inline bool Value::IsEmpty() const {
260              2  return _value == nullptr;
261               }
262
263            243inline napi_valuetype Value::Type() const {
264    ✗✓      243  if (_value == nullptr) {
265                   return napi_undefined;
266                 }
267
268                 napi_valuetype type;
269    ✓✗      243  napi_status status = napi_typeof(_env, _value, &type);
270 ✗✓✗✗    243  NAPI_THROW_IF_FAILED(_env, status, napi_undefined);
271            243  return type;
272               }
273
274             68inline bool Value::IsUndefined() const {
275             68  return Type() == napi_undefined;
276               }
277
278             14inline bool Value::IsNull() const {
279             14  return Type() == napi_null;
280               }
281
282             14inline bool Value::IsBoolean() const {
283             14  return Type() == napi_boolean;
284               }
285
286             32inline bool Value::IsNumber() const {
287             32  return Type() == napi_number;
288               }
```

```
289
290             44 inline bool Value::IsString() const {
291             44   return Type() == napi_string;
292                }
293
294             14 inline bool Value::IsSymbol() const {
295             14   return Type() == napi_symbol;
296                }
297
298             14 inline bool Value::IsArray() const {
299     ✗✓      14   if (_value == nullptr) {
300                    return false;
301                  }
302
303                  bool result;
304     ✓✗      14   napi_status status = napi_is_array(_env, _value, &result);
305   ✗✓✗✗      14   NAPI_THROW_IF_FAILED(_env, status, false);
306             14   return result;
307                }
308
309             19 inline bool Value::IsArrayBuffer() const {
310     ✗✓      19   if (_value == nullptr) {
311                    return false;
312                  }
313
314                  bool result;
315     ✓✗      19   napi_status status = napi_is_arraybuffer(_env, _value, &result);
316   ✗✓✗✗      19   NAPI_THROW_IF_FAILED(_env, status, false);
317             19   return result;
318                }
319
320             14 inline bool Value::IsTypedArray() const {
321     ✗✓      14   if (_value == nullptr) {
322                    return false;
323                  }
324
325                  bool result;
326     ✓✗      14   napi_status status = napi_is_typedarray(_env, _value, &result);
327   ✗✓✗✗      14   NAPI_THROW_IF_FAILED(_env, status, false);
328             14   return result;
329                }
330
331             14 inline bool Value::IsObject() const {
332   ✓✓✓✓      14   return Type() == napi_object || IsFunction();
333                }
334
335             22 inline bool Value::IsFunction() const {
336             22   return Type() == napi_function;
337                }
338
339             17 inline bool Value::IsPromise() const {
340     ✗✓      17   if (_value == nullptr) {
341                    return false;
342                  }
343
344                  bool result;
345     ✓✗      17   napi_status status = napi_is_promise(_env, _value, &result);
346   ✗✓✗✗      17   NAPI_THROW_IF_FAILED(_env, status, false);
347             17   return result;
348                }
349
350                #if NAPI_DATA_VIEW_FEATURE
351             14 inline bool Value::IsDataView() const {
352     ✗✓      14   if (_value == nullptr) {
353                    return false;
354                  }
355
356                  bool result;
357     ✓✗      14   napi_status status = napi_is_dataview(_env, _value, &result);
358   ✗✓✗✗      14   NAPI_THROW_IF_FAILED(_env, status, false);
359             14   return result;
360                }
361                #endif
362
363              6 inline bool Value::IsBuffer() const {
364     ✗✓       6   if (_value == nullptr) {
```

```
365                   return false;
366                 }
367
368                 bool result;
369     ✓✗        6   napi_status status = napi_is_buffer(_env, _value, &result);
370   ✗✓✗✗        6   NAPI_THROW_IF_FAILED(_env, status, false);
371               6   return result;
372                 }
373
374              14 inline bool Value::IsExternal() const {
375              14   return Type() == napi_external;
376                 }
377
378                 template <typename T>
379            2142 inline T Value::As() const {
380            2142   return T(_env, _value);
381                 }
382
383              24 inline Boolean Value::ToBoolean() const {
384                   napi_value result;
385     ✓✗       24   napi_status status = napi_coerce_to_bool(_env, _value, &result);
386   ✗✓✗✗       24   NAPI_THROW_IF_FAILED(_env, status, Boolean());
387     ✓✗       24   return Boolean(_env, result);
388                 }
389
390              18 inline Number Value::ToNumber() const {
391                   napi_value result;
392     ✓✗       18   napi_status status = napi_coerce_to_number(_env, _value, &result);
393   ✗✓✗✗       18   NAPI_THROW_IF_FAILED(_env, status, Number());
394     ✓✗       18   return Number(_env, result);
395                 }
396
397              16 inline String Value::ToString() const {
398                   napi_value result;
399     ✓✗       16   napi_status status = napi_coerce_to_string(_env, _value, &result);
400   ✗✓✗✗       16   NAPI_THROW_IF_FAILED(_env, status, String());
401     ✓✗       16   return String(_env, result);
402                 }
403
404              16 inline Object Value::ToObject() const {
405                   napi_value result;
406     ✓✗       16   napi_status status = napi_coerce_to_object(_env, _value, &result);
407   ✗✓✗✗       16   NAPI_THROW_IF_FAILED(_env, status, Object());
408     ✓✗       16   return Object(_env, result);
409                 }
410
411                 ////////////////////////////////////////////////////////////////////////////////
412                 // Boolean class
413                 ////////////////////////////////////////////////////////////////////////////////
414
415             291 inline Boolean Boolean::New(napi_env env, bool val) {
416                   napi_value value;
417     ✓✗      291   napi_status status = napi_get_boolean(env, val, &value);
418   ✗✓✗✗      291   NAPI_THROW_IF_FAILED(env, status, Boolean());
419     ✓✗      291   return Boolean(env, value);
420                 }
421
422                 inline Boolean::Boolean() : Napi::Value() {
423                 }
424
425             323 inline Boolean::Boolean(napi_env env, napi_value value) : Napi::Value(env, value) {
426             323 }
427
428               8 inline Boolean::operator bool() const {
429               8   return Value();
430                 }
431
432               8 inline bool Boolean::Value() const {
433                   bool result;
434     ✓✗        8   napi_status status = napi_get_value_bool(_env, _value, &result);
435   ✗✓✗✗        8   NAPI_THROW_IF_FAILED(_env, status, false);
436               8   return result;
437                 }
438
439                 ////////////////////////////////////////////////////////////////////////////////
```

```
440                // Number class
441                ////////////////////////////////////////////////////////////////////////////////
442
443             1154 inline Number Number::New(napi_env env, double val) {
444                    napi_value value;
445      ✓✗     1154   napi_status status = napi_create_double(env, val, &value);
446   ✗✓✗✗      1154   NAPI_THROW_IF_FAILED(env, status, Number());
447      ✓✗     1154   return Number(env, value);
448                }
449
450                inline Number::Number() : Value() {
451                }
452
453             2433 inline Number::Number(napi_env env, napi_value value) : Value(env, value) {
454             2433 }
455
456                inline Number::operator int32_t() const {
457                  return Int32Value();
458                }
459
460               56 inline Number::operator uint32_t() const {
461               56   return Uint32Value();
462                  }
463
464                inline Number::operator int64_t() const {
465                  return Int64Value();
466                }
467
468                inline Number::operator float() const {
469                  return FloatValue();
470                }
471
472                inline Number::operator double() const {
473                  return DoubleValue();
474                }
475
476              418 inline int32_t Number::Int32Value() const {
477                    int32_t result;
478      ✓✗      418   napi_status status = napi_get_value_int32(_env, _value, &result);
479   ✗✓✗✗       418   NAPI_THROW_IF_FAILED(_env, status, 0);
480              418   return result;
481                 }
482
483              422 inline uint32_t Number::Uint32Value() const {
484                    uint32_t result;
485      ✓✗      422   napi_status status = napi_get_value_uint32(_env, _value, &result);
486   ✗✓✗✗       422   NAPI_THROW_IF_FAILED(_env, status, 0);
487              422   return result;
488                 }
489
490              397 inline int64_t Number::Int64Value() const {
491                    int64_t result;
492      ✓✗      397   napi_status status = napi_get_value_int64(_env, _value, &result);
493   ✗✓✗✗       397   NAPI_THROW_IF_FAILED(_env, status, 0);
494              397   return result;
495                 }
496
497                8 inline float Number::FloatValue() const {
498                8   return static_cast<float>(DoubleValue());
499                  }
500
501               16 inline double Number::DoubleValue() const {
502                    double result;
503      ✓✗       16   napi_status status = napi_get_value_double(_env, _value, &result);
504   ✗✓✗✗       16   NAPI_THROW_IF_FAILED(_env, status, 0);
505               16   return result;
506                  }
507
508                ////////////////////////////////////////////////////////////////////////////////
509                // Name class
510                ////////////////////////////////////////////////////////////////////////////////
511
512                inline Name::Name() : Value() {
513                }
514
515          1000771 inline Name::Name(napi_env env, napi_value value) : Value(env, value) {
516          1000771 }
517
```

```
518                ////////////////////////////////////////////////////////////////////////////////
519                // String class
520                ////////////////////////////////////////////////////////////////////////////////
521
522              19 inline String String::New(napi_env env, const std::string& val) {
523              19   return String::New(env, val.c_str(), val.size());
524                 }
525
526               1 inline String String::New(napi_env env, const std::u16string& val) {
527               1   return String::New(env, val.c_str(), val.size());
528                 }
529
530         1000206 inline String String::New(napi_env env, const char* val) {
531                     napi_value value;
532    ✓✗   1000206   napi_status status = napi_create_string_utf8(env, val, std::strlen(val), &value);
533 ✗✓✗✗    1000206   NAPI_THROW_IF_FAILED(env, status, String());
534    ✓✗   1000206   return String(env, value);
535                 }
536
537              56 inline String String::New(napi_env env, const char16_t* val) {
538                     napi_value value;
539 ✓✗✓✗       56   napi_status status = napi_create_string_utf16(env, val, std::u16string(val).size(), &value);
540 ✗✓✗✗       56   NAPI_THROW_IF_FAILED(env, status, String());
541    ✓✗       56   return String(env, value);
542                 }
543
544              20 inline String String::New(napi_env env, const char* val, size_t length) {
545                     napi_value value;
546    ✓✗       20   napi_status status = napi_create_string_utf8(env, val, length, &value);
547 ✗✓✗✗       20   NAPI_THROW_IF_FAILED(env, status, String());
548    ✓✗       20   return String(env, value);
549                 }
550
551               2 inline String String::New(napi_env env, const char16_t* val, size_t length) {
552                     napi_value value;
553    ✓✗        2   napi_status status = napi_create_string_utf16(env, val, length, &value);
554 ✗✓✗✗        2   NAPI_THROW_IF_FAILED(env, status, String());
555    ✓✗        2   return String(env, value);
556                 }
557
558                 inline String::String() : Name() {
559                 }
560
561         1000735 inline String::String(napi_env env, napi_value value) : Name(env, value) {
562         1000735 }
563
564              47 inline String::operator std::string() const {
565              47   return Utf8Value();
566                 }
567
568               6 inline String::operator std::u16string() const {
569               6   return Utf16Value();
570                 }
571
572             347 inline std::string String::Utf8Value() const {
573                     size_t length;
574    ✓✗      347   napi_status status = napi_get_value_string_utf8(_env, _value, nullptr, 0, &length);
575 ✗✓✗✗       347   NAPI_THROW_IF_FAILED(_env, status, "");
576
577    ✓✗      347   std::string value;
578    ✓✗      347   value.reserve(length + 1);
579    ✓✗      347   value.resize(length);
580 ✓✗✓✗       347   status = napi_get_value_string_utf8(_env, _value, &value[0], value.capacity(), nullptr);
581 ✗✓✗✗       347   NAPI_THROW_IF_FAILED(_env, status, "");
582             347   return value;
583                 }
584
585              59 inline std::u16string String::Utf16Value() const {
586                     size_t length;
587    ✓✗       59   napi_status status = napi_get_value_string_utf16(_env, _value, nullptr, 0, &length);
588 ✗✓✗✗       59   NAPI_THROW_IF_FAILED(_env, status, NAPI_WIDE_TEXT(""));
589
590    ✓✗       59   std::u16string value;
591    ✓✗       59   value.reserve(length + 1);
```

```
592    ✓✗      59  value.resize(length);
593 ✓✗✓✗       59  status = napi_get_value_string_utf16(_env, _value, &value[0], value.capacity(), nullptr);
594 ✗✓✗✗       59  NAPI_THROW_IF_FAILED(_env, status, NAPI_WIDE_TEXT(""));
595             59  return value;
596               }
597
598               //////////////////////////////////////////////////////////////////////////////
599               // Symbol class
600               //////////////////////////////////////////////////////////////////////////////
601
602              1inline Symbol Symbol::New(napi_env env, const char* description) {
603                 napi_value descriptionValue = description != nullptr ?
    ✗✓✗✗
604 ✗✗✗✓        1    String::New(env, description) : static_cast<napi_value>(nullptr);
       ✗✗
605              1  return Symbol::New(env, descriptionValue);
606               }
607
608               inline Symbol Symbol::New(napi_env env, const std::string& description) {
609                 napi_value descriptionValue = String::New(env, description);
610                 return Symbol::New(env, descriptionValue);
611               }
612
613              1inline Symbol Symbol::New(napi_env env, String description) {
614              1  napi_value descriptionValue = description;
615              1  return Symbol::New(env, descriptionValue);
616               }
617
618              2inline Symbol Symbol::New(napi_env env, napi_value description) {
619                 napi_value value;
620    ✓✗        2  napi_status status = napi_create_symbol(env, description, &value);
621 ✗✓✗✗        2  NAPI_THROW_IF_FAILED(env, status, Symbol());
622    ✓✗        2  return Symbol(env, value);
623               }
624
625              1inline Symbol Symbol::WellKnown(napi_env env, const std::string& name) {
    ✓✗✓✗
626 ✓✗✓✗        1  return Napi::Env(env).Global().Get("Symbol").As<Object>().Get(name).As<Symbol>();
       ✓✗
627               }
628
629               inline Symbol::Symbol() : Name() {
630               }
631
632              3inline Symbol::Symbol(napi_env env, napi_value value) : Name(env, value) {
633              3}
634
635               //////////////////////////////////////////////////////////////////////////////
636               // Automagic value creation
637               //////////////////////////////////////////////////////////////////////////////
638
639               namespace details {
640               template <typename T>
641               struct vf_number {
642              5  static Number From(napi_env env, T value) {
643              5    return Number::New(env, static_cast<double>(value));
644                 }
645               };
646
647               template<>
648               struct vf_number<bool> {
649              4  static Boolean From(napi_env env, bool value) {
650              4    return Boolean::New(env, value);
651                 }
652               };
653
654               struct vf_utf8_charp {
655              1  static String From(napi_env env, const char* value) {
656              1    return String::New(env, value);
657                 }
658               };
659
660               struct vf_utf16_charp {
661              2  static String From(napi_env env, const char16_t* value) {
662              2    return String::New(env, value);
663                 }
664               };
665               struct vf_utf8_string {
```

```
666              14  static String From(napi_env env, const std::string& value) {
667              14    return String::New(env, value);
668                  }
669                };
670
671                struct vf_utf16_string {
672               1  static String From(napi_env env, const std::u16string& value) {
673               1    return String::New(env, value);
674                  }
675                };
676
677                template <typename T>
678                struct vf_fallback {
679             245  static Value From(napi_env env, const T& value) {
680             245    return Value(env, value);
681                  }
682                };
683
684                template <typename...> struct disjunction : std::false_type {};
685                template <typename B> struct disjunction<B> : B {};
686                template <typename B, typename... Bs>
687                struct disjunction<B, Bs...>
688                    : std::conditional<bool(B::value), B, disjunction<Bs...>>::type {};
689
690                template <typename T>
691                struct can_make_string
692                    : disjunction<typename std::is_convertible<T, const char *>::type,
693                                  typename std::is_convertible<T, const char16_t *>::type,
694                                  typename std::is_convertible<T, std::string>::type,
695                                  typename std::is_convertible<T, std::u16string>::type> {};
696                }
697
698                template <typename T>
699             272Value Value::From(napi_env env, const T& value) {
700                  using Helper = typename std::conditional<
701                    std::is_integral<T>::value || std::is_floating_point<T>::value,
702                    details::vf_number<T>,
703                    typename std::conditional<
704                      details::can_make_string<T>::value,
705                      String,
706                      details::vf_fallback<T>
707                    >::type
708                  >::type;
709             272  return Helper::From(env, value);
710                }
711
712                template <typename T>
713              18String String::From(napi_env env, const T& value) {
714                  struct Dummy {};
715                  using Helper = typename std::conditional<
716                    std::is_convertible<T, const char*>::value,
717                    details::vf_utf8_charp,
718                    typename std::conditional<
719                      std::is_convertible<T, const char16_t*>::value,
720                      details::vf_utf16_charp,
721                      typename std::conditional<
722                        std::is_convertible<T, std::string>::value,
723                        details::vf_utf8_string,
724                        typename std::conditional<
725                          std::is_convertible<T, std::u16string>::value,
726                          details::vf_utf16_string,
727                          Dummy
728                        >::type
729                      >::type
730                    >::type
731                  >::type;
732    ✓✗        18  return Helper::From(env, value);
733                }
734
735                //////////////////////////////////////////////////////////////////////////////
736                // Object class
737                //////////////////////////////////////////////////////////////////////////////
738
739                template <typename Key>
740                inline Object::PropertyLValue<Key>::operator Value() const {
741                  return Object(_env, _object).Get(_key);
742                }
743
744                template <typename Key> template <typename ValueType>
745             173inline Object::PropertyLValue<Key>& Object::PropertyLValue<Key>::operator =(ValueType value) {
746  ✓✗✓✗       173  Object(_env, _object).Set(_key, value);
```

```
✓✗✓✗
✓✗✓✗
✓✗✓✗
✓✗✓✗
   ✓✗
747            173  return *this;
748               }
749
750               template <typename Key>
751            173 inline Object::PropertyLValue<Key>::PropertyLValue(Object object, Key key)
752 ✗✓✗✗       173  : _env(object.Env()), _object(object), _key(key) {}
753
754             82 inline Object Object::New(napi_env env) {
755                  napi_value value;
756    ✓✗        82  napi_status status = napi_create_object(env, &value);
757 ✗✓✗✗        82  NAPI_THROW_IF_FAILED(env, status, Object());
758    ✓✗        82  return Object(env, value);
759               }
760
761              1 inline Object::Object() : Value() {
762              1 }
763
764           1137 inline Object::Object(napi_env env, napi_value value) : Value(env, value) {
765           1137 }
766
767            165 inline Object::PropertyLValue<std::string> Object::operator [](const char* utf8name) {
768 ✓✗✓✗       165  return PropertyLValue<std::string>(*this, utf8name);
769               }
770
771              7 inline Object::PropertyLValue<std::string> Object::operator [](const std::string& utf8name) {
772    ✓✗        7  return PropertyLValue<std::string>(*this, utf8name);
773               }
774
775              1 inline Object::PropertyLValue<uint32_t> Object::operator [](uint32_t index) {
776              1  return PropertyLValue<uint32_t>(*this, index);
777               }
778
779               inline Value Object::operator [](const char* utf8name) const {
780                 return Get(utf8name);
781               }
782
783               inline Value Object::operator [](const std::string& utf8name) const {
784                 return Get(utf8name);
785               }
786
787               inline Value Object::operator [](uint32_t index) const {
788                 return Get(index);
789               }
790
791              4 inline bool Object::Has(napi_value key) const {
792                  bool result;
793    ✓✗        4  napi_status status = napi_has_property(_env, _value, key, &result);
794 ✓✓✓✗        4  NAPI_THROW_IF_FAILED(_env, status, false);
795              3  return result;
796               }
797
798              4 inline bool Object::Has(Value key) const {
799                  bool result;
800 ✓✗✓✗        4  napi_status status = napi_has_property(_env, _value, key, &result);
801 ✓✓✓✗        4  NAPI_THROW_IF_FAILED(_env, status, false);
802              3  return result;
803               }
804
805              8 inline bool Object::Has(const char* utf8name) const {
806                  bool result;
807    ✓✗        8  napi_status status = napi_has_named_property(_env, _value, utf8name, &result);
808 ✓✓✓✗        8  NAPI_THROW_IF_FAILED(_env, status, false);
809              6  return result;
810               }
811
812              4 inline bool Object::Has(const std::string& utf8name) const {
813              4  return Has(utf8name.c_str());
814               }
815
816             12 inline bool Object::HasOwnProperty(napi_value key) const {
817                  bool result;
```

```
818    ✓✗      12  napi_status status = napi_has_own_property(_env, _value, key, &result);
819 ✓✓✓✗      12  NAPI_THROW_IF_FAILED(_env, status, false);
820              9  return result;
821               }
822
823              4 inline bool Object::HasOwnProperty(Value key) const {
824                  bool result;
825 ✓✗✓✗       4  napi_status status = napi_has_own_property(_env, _value, key, &result);
826 ✓✓✓✗       4  NAPI_THROW_IF_FAILED(_env, status, false);
827              3  return result;
828               }
829
830              8 inline bool Object::HasOwnProperty(const char* utf8name) const {
831                  napi_value key;
832    ✓✗        8  napi_status status = napi_create_string_utf8(_env, utf8name, std::strlen(utf8name), &key);
833 ✗✓✗✗       8  NAPI_THROW_IF_FAILED(_env, status, false);
834    ✓✓        8  return HasOwnProperty(key);
835               }
836
837              4 inline bool Object::HasOwnProperty(const std::string& utf8name) const {
838              4  return HasOwnProperty(utf8name.c_str());
839               }
840
841              2 inline Value Object::Get(napi_value key) const {
842                  napi_value result;
843    ✓✗        2  napi_status status = napi_get_property(_env, _value, key, &result);
844 ✓✓✓✗       2  NAPI_THROW_IF_FAILED(_env, status, Value());
845    ✓✗        1  return Value(_env, result);
846               }
847
848              2 inline Value Object::Get(Value key) const {
849                  napi_value result;
850 ✓✗✓✗       2  napi_status status = napi_get_property(_env, _value, key, &result);
851 ✓✓✓✗       2  NAPI_THROW_IF_FAILED(_env, status, Value());
852    ✓✗        1  return Value(_env, result);
853               }
854
855             10 inline Value Object::Get(const char* utf8name) const {
856                  napi_value result;
857    ✓✗       10  napi_status status = napi_get_named_property(_env, _value, utf8name, &result);
858 ✓✓✓✗      10  NAPI_THROW_IF_FAILED(_env, status, Value());
859    ✓✗        8  return Value(_env, result);
860               }
861
862              6 inline Value Object::Get(const std::string& utf8name) const {
863              6  return Get(utf8name.c_str());
864               }
865
866               template <typename ValueType>
867              2 inline void Object::Set(napi_value key, const ValueType& value) {
868                  napi_status status =
869 ✓✗✓✗       2      napi_set_property(_env, _value, key, Value::From(_env, value));
870 ✓✓✓✗       2  NAPI_THROW_IF_FAILED(_env, status);
871              1 }
872
873               template <typename ValueType>
874              2 inline void Object::Set(Value key, const ValueType& value) {
875                  napi_status status =
876 ✓✗✓✗       2      napi_set_property(_env, _value, key, Value::From(_env, value));
    ✓✗
877 ✓✓✓✗       2  NAPI_THROW_IF_FAILED(_env, status);
878              1 }
879
880               template <typename ValueType>
881            231 inline void Object::Set(const char* utf8name, const ValueType& value) {
882                  napi_status status =
883 ✓✗✓✗     231      napi_set_named_property(_env, _value, utf8name, Value::From(_env, value));
    ✓✗✓✗
    ✓✗✓✗
    ✓✗✓✗
    ✓✗✓✗
    ✓✗✓✗
    ✓✗✓✗
```

```
✓✗✓✗
✓✗✓✗
✓✗✓✗
✗✓✗✗
✓✓✓✗
✗✓✗✗
✗✓✗✗
✗✓✗✗
884 ✗✓✗✗      231  NAPI_THROW_IF_FAILED(_env, status);
✗✓✗✗
✗✓✗✗
✗✓✗✗
✗✓✗✗
885            229}
886
887               template <typename ValueType>
888            201inline void Object::Set(const std::string& utf8name, const ValueType& value) {
889            201  Set(utf8name.c_str(), value);
890            200}
891
892             4inline bool Object::Delete(napi_value key) {
893                bool result;
894    ✓✗       4  napi_status status = napi_delete_property(_env, _value, key, &result);
895 ✓✓✓✗       4  NAPI_THROW_IF_FAILED(_env, status, false);
896              3  return result;
897               }
898
899            12inline bool Object::Delete(Value key) {
900                bool result;
901 ✓✗✓✗       12  napi_status status = napi_delete_property(_env, _value, key, &result);
902 ✓✓✓✗       12  NAPI_THROW_IF_FAILED(_env, status, false);
903             9  return result;
904               }
905
906             4inline bool Object::Delete(const char* utf8name) {
907    ✓✓        4  return Delete(String::New(_env, utf8name));
908               }
909
910             4inline bool Object::Delete(const std::string& utf8name) {
911    ✓✓        4  return Delete(String::New(_env, utf8name));
912               }
913
914               inline bool Object::Has(uint32_t index) const {
915                 bool result;
916                 napi_status status = napi_has_element(_env, _value, index, &result);
917                 NAPI_THROW_IF_FAILED(_env, status, false);
918                 return result;
919               }
920
921            18inline Value Object::Get(uint32_t index) const {
922                napi_value value;
923    ✓✗      18  napi_status status = napi_get_element(_env, _value, index, &value);
924 ✗✓✗✗       18  NAPI_THROW_IF_FAILED(_env, status, Value());
925    ✓✗      18  return Value(_env, value);
926               }
927
928               template <typename ValueType>
929            37inline void Object::Set(uint32_t index, const ValueType& value) {
930                napi_status status =
931 ✓✗✓✗       37       napi_set_element(_env, _value, index, Value::From(_env, value));
✓✗✓✗
932 ✗✓✗✗       37  NAPI_THROW_IF_FAILED(_env, status);
✗✓✗✗
933            37}
934
935               inline bool Object::Delete(uint32_t index) {
936                 bool result;
937                 napi_status status = napi_delete_element(_env, _value, index, &result);
938                 NAPI_THROW_IF_FAILED(_env, status, false);
939                 return result;
940               }
941
942             1inline Array Object::GetPropertyNames() {
943                napi_value result;
```

```
944     ✓✗       1  napi_status status = napi_get_property_names(_env, _value, &result);
945  ✗✓✗✗        1  NAPI_THROW_IF_FAILED(_env, status, Array());
946     ✓✗       1  return Array(_env, result);
947               }
948
949              1inline void Object::DefineProperty(const PropertyDescriptor& property) {
950                 napi_status status = napi_define_properties(_env, _value, 1,
951              1    reinterpret_cast<const napi_property_descriptor*>(&property));
952  ✗✓✗✗        1  NAPI_THROW_IF_FAILED(_env, status);
953              1}
954
955              3inline void Object::DefineProperties(const std::initializer_list<PropertyDescriptor>& properties) {
956                 napi_status status = napi_define_properties(_env, _value, properties.size(),
957              3    reinterpret_cast<const napi_property_descriptor*>(properties.begin()));
958  ✗✓✗✗        3  NAPI_THROW_IF_FAILED(_env, status);
959              3}
960
961               inline void Object::DefineProperties(const std::vector<PropertyDescriptor>& properties) {
962                 napi_status status = napi_define_properties(_env, _value, properties.size(),
963                   reinterpret_cast<const napi_property_descriptor*>(properties.data()));
964                 NAPI_THROW_IF_FAILED(_env, status);
965               }
966
967               inline bool Object::InstanceOf(const Function& constructor) const {
968                 bool result;
969                 napi_status status = napi_instanceof(_env, _value, constructor, &result);
970                 NAPI_THROW_IF_FAILED(_env, status, false);
971                 return result;
972               }
973
974               ////////////////////////////////////////////////////////////////////////////////
975               // External class
976               ////////////////////////////////////////////////////////////////////////////////
977
978               template <typename T>
979              2inline External<T> External<T>::New(napi_env env, T* data) {
980                 napi_value value;
981     ✓✗       2  napi_status status = napi_create_external(env, data, nullptr, nullptr, &value);
982  ✗✓✗✗        2  NAPI_THROW_IF_FAILED(env, status, External());
983     ✓✗       2  return External(env, value);
984               }
985
986               template <typename T>
987               template <typename Finalizer>
988              1inline External<T> External<T>::New(napi_env env,
989                                                   T* data,
990                                                   Finalizer finalizeCallback) {
991                 napi_value value;
992                 details::FinalizeData<T, Finalizer>* finalizeData =
993     ✓✗       1    new details::FinalizeData<T, Finalizer>({ finalizeCallback, nullptr });
994                 napi_status status = napi_create_external(
995                   env,
996                   data,
997                   details::FinalizeData<T, Finalizer>::Wrapper,
998                   finalizeData,
999     ✓✗       1    &value);
1000    ✗✓       1  if (status != napi_ok) {
1001                   delete finalizeData;
1002                   NAPI_THROW_IF_FAILED(env, status, External());
1003                 }
1004    ✓✗       1  return External(env, value);
1005              }
1006
1007              template <typename T>
1008              template <typename Finalizer, typename Hint>
1009             1inline External<T> External<T>::New(napi_env env,
1010                                                  T* data,
1011                                                  Finalizer finalizeCallback,
1012                                                  Hint* finalizeHint) {
1013                napi_value value;
1014                details::FinalizeData<T, Finalizer, Hint>* finalizeData =
1015    ✓✗       1    new details::FinalizeData<T, Finalizer, Hint>({ finalizeCallback, finalizeHint });
1016                napi_status status = napi_create_external(
1017                  env,
1018                  data,
1019                  details::FinalizeData<T, Finalizer, Hint>::WrapperWithHint,
1020                  finalizeData,
```

```
1021    ✓✗       1    &value);
1022    ✗✓       1  if (status != napi_ok) {
1023                   delete finalizeData;
1024                   NAPI_THROW_IF_FAILED(env, status, External());
1025                 }
1026    ✓✗       1  return External(env, value);
1027                }
1028
1029                template <typename T>
1030                inline External<T>::External() : Value() {
1031                }
1032
1033                template <typename T>
1034               7inline External<T>::External(napi_env env, napi_value value) : Value(env, value) {
1035               7}
1036
1037                template <typename T>
1038               3inline T* External<T>::Data() const {
1039                  void* data;
1040    ✓✗       3  napi_status status = napi_get_value_external(_env, _value, &data);
1041 ✗✓✗✗       3  NAPI_THROW_IF_FAILED(_env, status, nullptr);
1042               3  return reinterpret_cast<T*>(data);
1043                }
1044
1045                ////////////////////////////////////////////////////////////////////////////////
1046                // Array class
1047                ////////////////////////////////////////////////////////////////////////////////
1048
1049               3inline Array Array::New(napi_env env) {
1050                  napi_value value;
1051    ✓✗       3  napi_status status = napi_create_array(env, &value);
1052 ✗✓✗✗       3  NAPI_THROW_IF_FAILED(env, status, Array());
1053    ✓✗       3  return Array(env, value);
1054                }
1055
1056                inline Array Array::New(napi_env env, size_t length) {
1057                  napi_value value;
1058                  napi_status status = napi_create_array_with_length(env, length, &value);
1059                  NAPI_THROW_IF_FAILED(env, status, Array());
1060                  return Array(env, value);
1061                }
1062
1063                inline Array::Array() : Object() {
1064                }
1065
1066               4inline Array::Array(napi_env env, napi_value value) : Object(env, value) {
1067               4}
1068
1069                inline uint32_t Array::Length() const {
1070                  uint32_t result;
1071                  napi_status status = napi_get_array_length(_env, _value, &result);
1072                  NAPI_THROW_IF_FAILED(_env, status, 0);
1073                  return result;
1074                }
1075
1076                ////////////////////////////////////////////////////////////////////////////////
1077                // ArrayBuffer class
1078                ////////////////////////////////////////////////////////////////////////////////
1079
1080              10inline ArrayBuffer ArrayBuffer::New(napi_env env, size_t byteLength) {
1081                  napi_value value;
1082                  void* data;
1083    ✓✗      10  napi_status status = napi_create_arraybuffer(env, byteLength, &data, &value);
1084 ✗✓✗✗      10  NAPI_THROW_IF_FAILED(env, status, ArrayBuffer());
1085
1086    ✓✗      10  return ArrayBuffer(env, value, data, byteLength);
1087                }
1088
1089               1inline ArrayBuffer ArrayBuffer::New(napi_env env,
1090                                                    void* externalData,
1091                                                    size_t byteLength) {
1092                  napi_value value;
1093                  napi_status status = napi_create_external_arraybuffer(
1094    ✓✗       1    env, externalData, byteLength, nullptr, nullptr, &value);
1095 ✗✓✗✗       1  NAPI_THROW_IF_FAILED(env, status, ArrayBuffer());
1096
1097    ✓✗       1  return ArrayBuffer(env, value, externalData, byteLength);
```

```
1098              }
1099
1100              template <typename Finalizer>
1101             1inline ArrayBuffer ArrayBuffer::New(napi_env env,
1102                                                  void* externalData,
1103                                                  size_t byteLength,
1104                                                  Finalizer finalizeCallback) {
1105                napi_value value;
1106                details::FinalizeData<void, Finalizer>* finalizeData =
1107    ✓✗          1    new details::FinalizeData<void, Finalizer>({ finalizeCallback, nullptr });
1108                napi_status status = napi_create_external_arraybuffer(
1109                  env,
1110                  externalData,
1111                  byteLength,
1112                  details::FinalizeData<void, Finalizer>::Wrapper,
1113                  finalizeData,
1114    ✓✗          1    &value);
1115    ✗✓          1  if (status != napi_ok) {
1116                    delete finalizeData;
1117                    NAPI_THROW_IF_FAILED(env, status, ArrayBuffer());
1118                }
1119
1120    ✓✗          1  return ArrayBuffer(env, value, externalData, byteLength);
1121              }
1122
1123              template <typename Finalizer, typename Hint>
1124             1inline ArrayBuffer ArrayBuffer::New(napi_env env,
1125                                                  void* externalData,
1126                                                  size_t byteLength,
1127                                                  Finalizer finalizeCallback,
1128                                                  Hint* finalizeHint) {
1129                napi_value value;
1130                details::FinalizeData<void, Finalizer, Hint>* finalizeData =
1131    ✓✗          1    new details::FinalizeData<void, Finalizer, Hint>({ finalizeCallback, finalizeHint });
1132                napi_status status = napi_create_external_arraybuffer(
1133                  env,
1134                  externalData,
1135                  byteLength,
1136                  details::FinalizeData<void, Finalizer, Hint>::WrapperWithHint,
1137                  finalizeData,
1138    ✓✗          1    &value);
1139    ✗✓          1  if (status != napi_ok) {
1140                    delete finalizeData;
1141                    NAPI_THROW_IF_FAILED(env, status, ArrayBuffer());
1142                }
1143
1144    ✓✗          1  return ArrayBuffer(env, value, externalData, byteLength);
1145              }
1146
1147             1inline ArrayBuffer::ArrayBuffer() : Object(), _data(nullptr), _length(0) {
1148             1}
1149
1150            59inline ArrayBuffer::ArrayBuffer(napi_env env, napi_value value)
1151            59  : Object(env, value), _data(nullptr), _length(0) {
1152            59}
1153
1154            13inline ArrayBuffer::ArrayBuffer(napi_env env, napi_value value, void* data, size_t length)
1155            13  : Object(env, value), _data(data), _length(length) {
1156            13}
1157
1158            27inline void* ArrayBuffer::Data() {
1159            27  EnsureInfo();
1160            27  return _data;
1161              }
1162
1163            22inline size_t ArrayBuffer::ByteLength() {
1164            22  EnsureInfo();
1165            22  return _length;
1166              }
1167
1168            49inline void ArrayBuffer::EnsureInfo() const {
1169                // The ArrayBuffer instance may have been constructed from a napi_value whose
1170                // length/data are not yet known. Fetch and cache these values just once,
1171                // since they can never change during the lifetime of the ArrayBuffer.
1172    ✓✓      49  if (_data == nullptr) {
1173            22    napi_status status = napi_get_arraybuffer_info(_env, _value, &_data, &_length);
1174 ✗✓✗✗       22    NAPI_THROW_IF_FAILED(_env, status);
1175                }
```

```
1176              49 }
1177
1178                 #if NAPI_DATA_VIEW_FEATURE
1179                 ////////////////////////////////////////////////////////////////////////////
1180                 // DataView class
1181                 ////////////////////////////////////////////////////////////////////////////
1182              1 inline DataView DataView::New(napi_env env,
1183                                              Napi::ArrayBuffer arrayBuffer) {
1184              1   return New(env, arrayBuffer, 0, arrayBuffer.ByteLength());
1185                  }
1186
1187              3 inline DataView DataView::New(napi_env env,
1188                                              Napi::ArrayBuffer arrayBuffer,
1189                                              size_t byteOffset) {
1190     ✓✓       3   if (byteOffset > arrayBuffer.ByteLength()) {
1191              1     NAPI_THROW(RangeError::New(env,
1192     ✓✗       1         "Start offset is outside the bounds of the buffer"));
1193                    return DataView();
1194                  }
1195                  return New(env, arrayBuffer, byteOffset,
1196              2       arrayBuffer.ByteLength() - byteOffset);
1197                  }
1198
1199              7 inline DataView DataView::New(napi_env env,
1200                                              Napi::ArrayBuffer arrayBuffer,
1201                                              size_t byteOffset,
1202                                              size_t byteLength) {
1203 ✓✗✓✓       7   if (byteOffset + byteLength > arrayBuffer.ByteLength()) {
1204     ✓✗       2     NAPI_THROW(RangeError::New(env, "Invalid DataView length"));
1205                    return DataView();
1206                  }
1207                  napi_value value;
1208                  napi_status status = napi_create_dataview(
1209 ✓✗✓✗       5     env, byteLength, arrayBuffer, byteOffset, &value);
1210 ✗✓✗✗       5   NAPI_THROW_IF_FAILED(env, status, DataView());
1211     ✓✗       5   return DataView(env, value);
1212                  }
1213
1214                  inline DataView::DataView() : Object() {
1215                  }
1216
1217             57 inline DataView::DataView(napi_env env, napi_value value) : Object(env, value) {
1218                  napi_status status = napi_get_dataview_info(
1219                    _env,
1220                    _value   /* dataView */,
1221                    &_length /* byteLength */,
1222                    &_data   /* data */,
1223                    nullptr  /* arrayBuffer */,
1224             57     nullptr  /* byteOffset */);
1225 ✗✓✗✗      57   NAPI_THROW_IF_FAILED(_env, status);
1226             57 }
1227
1228             10 inline Napi::ArrayBuffer DataView::ArrayBuffer() const {
1229                  napi_value arrayBuffer;
1230                  napi_status status = napi_get_dataview_info(
1231                    _env,
1232                    _value       /* dataView */,
1233                    nullptr      /* byteLength */,
1234                    nullptr      /* data */,
1235                    &arrayBuffer /* arrayBuffer */,
1236     ✓✗      10     nullptr      /* byteOffset */);
1237 ✗✓✗✗      10   NAPI_THROW_IF_FAILED(_env, status, Napi::ArrayBuffer());
1238     ✓✗      10   return Napi::ArrayBuffer(_env, arrayBuffer);
1239                  }
1240
1241              5 inline size_t DataView::ByteOffset() const {
1242                  size_t byteOffset;
1243                  napi_status status = napi_get_dataview_info(
1244                    _env,
1245                    _value      /* dataView */,
1246                    nullptr     /* byteLength */,
1247                    nullptr     /* data */,
1248                    nullptr     /* arrayBuffer */,
1249     ✓✗       5     &byteOffset /* byteOffset */);
1250 ✗✓✗✗       5   NAPI_THROW_IF_FAILED(_env, status, 0);
1251              5   return byteOffset;
1252                  }
```

```
1253
1254              5 inline size_t DataView::ByteLength() const {
1255              5   return _length;
1256                }
1257
1258                inline void* DataView::Data() const {
1259                  return _data;
1260                }
1261
1262              2 inline float DataView::GetFloat32(size_t byteOffset) const {
1263              2   return ReadData<float>(byteOffset);
1264                }
1265
1266              2 inline double DataView::GetFloat64(size_t byteOffset) const {
1267              2   return ReadData<double>(byteOffset);
1268                }
1269
1270              2 inline int8_t DataView::GetInt8(size_t byteOffset) const {
1271              2   return ReadData<int8_t>(byteOffset);
1272                }
1273
1274              2 inline int16_t DataView::GetInt16(size_t byteOffset) const {
1275              2   return ReadData<int16_t>(byteOffset);
1276                }
1277
1278              2 inline int32_t DataView::GetInt32(size_t byteOffset) const {
1279              2   return ReadData<int32_t>(byteOffset);
1280                }
1281
1282              2 inline uint8_t DataView::GetUint8(size_t byteOffset) const {
1283              2   return ReadData<uint8_t>(byteOffset);
1284                }
1285
1286              2 inline uint16_t DataView::GetUint16(size_t byteOffset) const {
1287              2   return ReadData<uint16_t>(byteOffset);
1288                }
1289
1290              2 inline uint32_t DataView::GetUint32(size_t byteOffset) const {
1291              2   return ReadData<uint32_t>(byteOffset);
1292                }
1293
1294              2 inline void DataView::SetFloat32(size_t byteOffset, float value) const {
1295              2   WriteData<float>(byteOffset, value);
1296              1 }
1297
1298              2 inline void DataView::SetFloat64(size_t byteOffset, double value) const {
1299              2   WriteData<double>(byteOffset, value);
1300              1 }
1301
1302              2 inline void DataView::SetInt8(size_t byteOffset, int8_t value) const {
1303              2   WriteData<int8_t>(byteOffset, value);
1304              1 }
1305
1306              2 inline void DataView::SetInt16(size_t byteOffset, int16_t value) const {
1307              2   WriteData<int16_t>(byteOffset, value);
1308              1 }
1309
1310              2 inline void DataView::SetInt32(size_t byteOffset, int32_t value) const {
1311              2   WriteData<int32_t>(byteOffset, value);
1312              1 }
1313
1314              2 inline void DataView::SetUint8(size_t byteOffset, uint8_t value) const {
1315              2   WriteData<uint8_t>(byteOffset, value);
1316              1 }
1317
1318              2 inline void DataView::SetUint16(size_t byteOffset, uint16_t value) const {
1319              2   WriteData<uint16_t>(byteOffset, value);
1320              1 }
1321
1322              2 inline void DataView::SetUint32(size_t byteOffset, uint32_t value) const {
1323              2   WriteData<uint32_t>(byteOffset, value);
1324              1 }
1325
1326                template <typename T>
1327             16 inline T DataView::ReadData(size_t byteOffset) const {
1328 ✓✓✗✓        16   if (byteOffset + sizeof(T) > _length ||
     ✓✓✗✓
     ✓✓✗✓
     ✓✓✗✓
     ✓✓✗✓
```

```
✓✓✗✓
✓✓✗✓
✓✓✗✓
1329                      byteOffset + sizeof(T) < byteOffset) {  // overflow
✓✗✓✗
✓✗✓✗
1330          8    NAPI_THROW(RangeError::New(_env,
✓✗✓✗
✓✗✓✗
1331                         "Offset is outside the bounds of the DataView"));
1332                   return 0;
1333                 }
1334
1335                 8  return *reinterpret_cast<T*>(static_cast<uint8_t*>(_data) + byteOffset);
1336                  }
1337
1338                  template <typename T>
1339                16 inline void DataView::WriteData(size_t byteOffset, T value) const {
✓✓✗✓
✓✓✗✓
✓✓✗✓
✓✓✗✓
1340          16    16  if (byteOffset + sizeof(T) > _length ||
✓✓✗✓
✓✓✗✓
✓✓✗✓
✓✓✗✓
1341                      byteOffset + sizeof(T) < byteOffset) {  // overflow
✓✗✓✗
✓✗✓✗
1342          8    NAPI_THROW(RangeError::New(_env,
✓✗✓✗
✓✗✓✗
1343                         "Offset is outside the bounds of the DataView"));
1344                   return;
1345                 }
1346
1347                 8  *reinterpret_cast<T*>(static_cast<uint8_t*>(_data) + byteOffset) = value;
1348                 8 }
1349                  #endif
1350
1351                  ////////////////////////////////////////////////////////////////////////////////
1352                  // TypedArray class
1353                  ////////////////////////////////////////////////////////////////////////////////
1354
1355                  inline TypedArray::TypedArray()
1356                    : Object(), _type(TypedArray::unknown_array_type), _length(0) {
1357                  }
1358
1359               173 inline TypedArray::TypedArray(napi_env env, napi_value value)
1360               173   : Object(env, value), _type(TypedArray::unknown_array_type), _length(0) {
1361               173 }
1362
1363                18 inline TypedArray::TypedArray(napi_env env,
1364                                                napi_value value,
1365                                                napi_typedarray_type type,
1366                                                size_t length)
1367                18   : Object(env, value), _type(type), _length(length) {
1368                18 }
1369
1370                72 inline napi_typedarray_type TypedArray::TypedArrayType() const {
1371    ✓✗          72   if (_type == TypedArray::unknown_array_type) {
1372                       napi_status status = napi_get_typedarray_info(_env, _value,
1373                         &const_cast<TypedArray*>(this)->_type, &const_cast<TypedArray*>(this)->_length,
1374                72       nullptr, nullptr, nullptr);
1375 ✗✓✗✗          72     NAPI_THROW_IF_FAILED(_env, status, napi_int8_array);
1376                      }
1377
1378                72   return _type;
1379                  }
1380
1381                  inline uint8_t TypedArray::ElementSize() const {
1382                    switch (TypedArrayType()) {
1383                      case napi_int8_array:
1384                      case napi_uint8_array:
1385                      case napi_uint8_clamped_array:
1386                        return 1;
1387                      case napi_int16_array:
1388                      case napi_uint16_array:
```

```
1389                     return 2;
1390                   case napi_int32_array:
1391                   case napi_uint32_array:
1392                   case napi_float32_array:
1393                     return 4;
1394                   case napi_float64_array:
1395                     return 8;
1396                   default:
1397                     return 0;
1398                 }
1399               }
1400
1401             18inline size_t TypedArray::ElementLength() const {
1402    ✓✗       18  if (_type == TypedArray::unknown_array_type) {
1403                   napi_status status = napi_get_typedarray_info(_env, _value,
1404                     &const_cast<TypedArray*>(this)->_type, &const_cast<TypedArray*>(this)->_length,
1405             18      nullptr, nullptr, nullptr);
1406 ✗✓✗✗      18    NAPI_THROW_IF_FAILED(_env, status, 0);
1407                 }
1408
1409             18  return _length;
1410               }
1411
1412               inline size_t TypedArray::ByteOffset() const {
1413                 size_t byteOffset;
1414                 napi_status status = napi_get_typedarray_info(
1415                   _env, _value, nullptr, nullptr, nullptr, nullptr, &byteOffset);
1416                 NAPI_THROW_IF_FAILED(_env, status, 0);
1417                 return byteOffset;
1418               }
1419
1420               inline size_t TypedArray::ByteLength() const {
1421                 return ElementSize() * ElementLength();
1422               }
1423
1424             18inline Napi::ArrayBuffer TypedArray::ArrayBuffer() const {
1425                 napi_value arrayBuffer;
1426                 napi_status status = napi_get_typedarray_info(
1427    ✓✗       18    _env, _value, nullptr, nullptr, nullptr, &arrayBuffer, nullptr);
1428 ✗✓✗✗      18  NAPI_THROW_IF_FAILED(_env, status, Napi::ArrayBuffer());
1429    ✓✗       18  return Napi::ArrayBuffer(_env, arrayBuffer);
1430               }
1431
1432               //////////////////////////////////////////////////////////////////////////////
1433               // TypedArrayOf<T> class
1434               //////////////////////////////////////////////////////////////////////////////
1435
1436               template <typename T>
1437              9inline TypedArrayOf<T> TypedArrayOf<T>::New(napi_env env,
1438                                                           size_t elementLength,
1439                                                           napi_typedarray_type type) {
     ✓✗✓✗
     ✓✗✓✗
1440  ✓✗✓✗       9  Napi::ArrayBuffer arrayBuffer = Napi::ArrayBuffer::New(env, elementLength * sizeof (T));
     ✓✗✓✗
     ✓✗✓✗
     ✓✗✓✗
     ✓✗✓✗
1441  ✓✗✓✗       9  return New(env, elementLength, arrayBuffer, 0, type);
     ✓✗✓✗
     ✓✗✓✗
1442               }
1443
1444               template <typename T>
1445             19inline TypedArrayOf<T> TypedArrayOf<T>::New(napi_env env,
1446                                                           size_t elementLength,
1447                                                           Napi::ArrayBuffer arrayBuffer,
1448                                                           size_t bufferOffset,
1449                                                           napi_typedarray_type type) {
1450                 napi_value value;
1451                 napi_status status = napi_create_typedarray(
1452  ✓✗✓✗      19    env, type, elementLength, arrayBuffer, bufferOffset, &value);
     ✓✗✓✗
     ✓✗✓✗
     ✓✗✓✗
     ✓✗✓✗
     ✓✗✓✗
```

```
✓✗✓✗
✓✗✓✗
✗✓✗✗
✗✓✗✗
✗✓✗✗
✗✓✗✗
1453                19  NAPI_THROW_IF_FAILED(env, status, TypedArrayOf<T>());
✗✓✗✗
✗✓✗✗
✗✓✗✗
✓✓✓✗
1454
1455                    return TypedArrayOf<T>(
1456                      env, value, type, elementLength,
✓✗✓✗
✓✗✓✗
✓✗✓✗
✓✗✓✗
1457                18    reinterpret_cast<T*>(reinterpret_cast<uint8_t*>(arrayBuffer.Data()) + bufferOffset));
✓✗✓✗
✓✗✓✗
✓✗✓✗
✓✗✓✗
1458                  }
1459
1460                  template <typename T>
1461                  inline TypedArrayOf<T>::TypedArrayOf() : TypedArray(), _data(nullptr) {
1462                  }
1463
1464                  template <typename T>
1465                65inline TypedArrayOf<T>::TypedArrayOf(napi_env env, napi_value value)
1466                65  : TypedArray(env, value), _data(nullptr) {
1467                    napi_status status = napi_get_typedarray_info(
1468                65    _env, _value, &_type, &_length, reinterpret_cast<void**>(&_data), nullptr, nullptr);
✗✓✗✗
✗✓✗✗
✗✓✗✗
✗✓✗✗
1469                65  NAPI_THROW_IF_FAILED(_env, status);
✗✓✗✗
✗✓✗✗
✗✗✗✗
✗✓✗✗
1470                65}
1471
1472                  template <typename T>
1473                18inline TypedArrayOf<T>::TypedArrayOf(napi_env env,
1474                                                      napi_value value,
1475                                                      napi_typedarray_type type,
1476                                                      size_t length,
1477                                                      T* data)
1478                18  : TypedArray(env, value, type, length), _data(data) {
✗✓✗✓
✗✓✗✓
✗✓✗✓
✗✓✗✓
1479 ✗✓✗✓      18  if (!(type == TypedArrayTypeForPrimitiveType<T>() ||
✗✓✗✓
✓✓✗✓
✗✓✗✓
   ✗✓
1480                      (type == napi_uint8_clamped_array && std::is_same<T, uint8_t>::value))) {
1481                    NAPI_THROW(TypeError::New(env, "Array type must match the template parameter. "
1482                      "(Uint8 arrays may optionally have the \"clamped\" array type.)"));
1483                  }
1484                18}
1485
1486                  template <typename T>
1487                54inline T& TypedArrayOf<T>::operator [](size_t index) {
1488                54  return _data[index];
1489                  }
1490
1491                  template <typename T>
1492                  inline const T& TypedArrayOf<T>::operator [](size_t index) const {
1493                    return _data[index];
1494                  }
```

```
1495
1496                template <typename T>
1497                inline T* TypedArrayOf<T>::Data() {
1498                  return _data;
1499                }
1500
1501                template <typename T>
1502                inline const T* TypedArrayOf<T>::Data() const {
1503                  return _data;
1504                }
1505
1506                ////////////////////////////////////////////////////////////////////////////////
1507                // Function class
1508                ////////////////////////////////////////////////////////////////////////////////
1509
1510                template <typename Callable>
1511              151inline Function Function::New(napi_env env,
1512                                             Callable cb,
1513                                             const char* utf8name,
1514                                             void* data) {
1515                  typedef decltype(cb(CallbackInfo(nullptr, nullptr))) ReturnType;
1516                  typedef details::CallbackData<Callable, ReturnType> CbData;
1517                  // TODO: Delete when the function is destroyed
1518 ✓✗✓✗ ✓✗    151  auto callbackData = new CbData({ cb, data });
1519
1520                  napi_value value;
1521                  napi_status status = napi_create_function(
1522 ✓✗✓✗ ✓✗    151    env, utf8name, NAPI_AUTO_LENGTH, CbData::Wrapper, callbackData, &value);
1523 ✗✓✗✗ ✗✓✗✗ ✗✓✗✗ 151  NAPI_THROW_IF_FAILED(env, status, Function());
1524 ✓✗✓✗ ✓✗    151  return Function(env, value);
1525                }
1526
1527                template <typename Callable>
1528                1inline Function Function::New(napi_env env,
1529                                             Callable cb,
1530                                             const std::string& utf8name,
1531                                             void* data) {
1532                1  return New(env, cb, utf8name.c_str(), data);
1533                 }
1534
1535                inline Function::Function() : Object() {
1536                }
1537
1538              178inline Function::Function(napi_env env, napi_value value) : Object(env, value) {
1539              178}
1540
1541                7inline Value Function::operator ()(const std::initializer_list<napi_value>& args) const {
1542 ✓✗✓✗ ✓✓     7  return Call(Env().Undefined(), args);
1543                 }
1544
1545                1inline Value Function::Call(const std::initializer_list<napi_value>& args) const {
1546 ✓✗✓✗ ✗✓     1  return Call(Env().Undefined(), args);
1547                 }
1548
1549                1inline Value Function::Call(const std::vector<napi_value>& args) const {
1550 ✓✗✓✗ ✓✗     1  return Call(Env().Undefined(), args);
1551                 }
1552
1553                inline Value Function::Call(size_t argc, const napi_value* args) const {
1554                  return Call(Env().Undefined(), argc, args);
1555                }
1556
1557               10inline Value Function::Call(napi_value recv, const std::initializer_list<napi_value>& args) const {
1558               10  return Call(recv, args.size(), args.begin());
1559                 }
1560
1561                2inline Value Function::Call(napi_value recv, const std::vector<napi_value>& args) const {
1562                2  return Call(recv, args.size(), args.data());
1563                 }
1564
```

```
1565              12 inline Value Function::Call(napi_value recv, size_t argc, const napi_value* args) const {
1566                     napi_value result;
1567                     napi_status status = napi_call_function(
1568     ✓✗       12       _env, recv, _value, argc, args, &result);
1569 ✓✓✓✗         12     NAPI_THROW_IF_FAILED(_env, status, Value());
1570     ✓✗           6    return Value(_env, result);
1571                    }
1572
1573               2 inline Value Function::MakeCallback(
1574                       napi_value recv, const std::initializer_list<napi_value>& args) const {
1575               2    return MakeCallback(recv, args.size(), args.begin());
1576                    }
1577
1578                    inline Value Function::MakeCallback(
1579                        napi_value recv, const std::vector<napi_value>& args) const {
1580                      return MakeCallback(recv, args.size(), args.data());
1581                    }
1582
1583               2 inline Value Function::MakeCallback(
1584                       napi_value recv, size_t argc, const napi_value* args) const {
1585                      napi_value result;
1586                      napi_status status = napi_make_callback(
1587     ✓✗           2      _env, nullptr, recv, _value, argc, args, &result);
1588 ✗✓✗✗         2    NAPI_THROW_IF_FAILED(_env, status, Value());
1589     ✓✗           2    return Value(_env, result);
1590                    }
1591
1592               5 inline Object Function::New(const std::initializer_list<napi_value>& args) const {
1593               5    return New(args.size(), args.begin());
1594                    }
1595
1596               1 inline Object Function::New(const std::vector<napi_value>& args) const {
1597               1    return New(args.size(), args.data());
1598                    }
1599
1600               6 inline Object Function::New(size_t argc, const napi_value* args) const {
1601                      napi_value result;
1602                      napi_status status = napi_new_instance(
1603     ✓✗           6      _env, _value, argc, args, &result);
1604 ✗✓✗✗         6    NAPI_THROW_IF_FAILED(_env, status, Object());
1605     ✓✗           6    return Object(_env, result);
1606                    }
1607
1608                    ////////////////////////////////////////////////////////////////////////////
1609                    // Promise class
1610                    ////////////////////////////////////////////////////////////////////////////
1611
1612               2 inline Promise::Deferred Promise::Deferred::New(napi_env env) {
1613               2    return Promise::Deferred(env);
1614                    }
1615
1616               2 inline Promise::Deferred::Deferred(napi_env env) : _env(env) {
1617               2    napi_status status = napi_create_promise(_env, &_deferred, &_promise);
1618 ✗✓✗✗         2    NAPI_THROW_IF_FAILED(_env, status);
1619               2 }
1620
1621               2 inline Promise Promise::Deferred::Promise() const {
1622               2    return Napi::Promise(_env, _promise);
1623                    }
1624
1625                    inline Napi::Env Promise::Deferred::Env() const {
1626                      return Napi::Env(_env);
1627                    }
1628
1629               1 inline void Promise::Deferred::Resolve(napi_value value) const {
1630               1    napi_status status = napi_resolve_deferred(_env, _deferred, value);
1631 ✗✓✗✗         1    NAPI_THROW_IF_FAILED(_env, status);
1632               1 }
1633
1634               1 inline void Promise::Deferred::Reject(napi_value value) const {
1635               1    napi_status status = napi_reject_deferred(_env, _deferred, value);
1636 ✗✓✗✗         1    NAPI_THROW_IF_FAILED(_env, status);
1637               1 }
1638
1639               2 inline Promise::Promise(napi_env env, napi_value value) : Object(env, value) {
1640               2 }
1641
1642                    ////////////////////////////////////////////////////////////////////////////
```

```
1643                // Buffer<T> class
1644                ////////////////////////////////////////////////////////////////////////////////
1645
1646                template <typename T>
1647               1inline Buffer<T> Buffer<T>::New(napi_env env, size_t length) {
1648                  napi_value value;
1649                  void* data;
1650    ✓✗        1  napi_status status = napi_create_buffer(env, length * sizeof (T), &data, &value);
1651 ✗✓✗✗       1  NAPI_THROW_IF_FAILED(env, status, Buffer<T>());
1652    ✓✗        1  return Buffer(env, value, length, static_cast<T*>(data));
1653                }
1654
1655                template <typename T>
1656               1inline Buffer<T> Buffer<T>::New(napi_env env, T* data, size_t length) {
1657                  napi_value value;
1658                  napi_status status = napi_create_external_buffer(
1659    ✓✗        1    env, length * sizeof (T), data, nullptr, nullptr, &value);
1660 ✗✓✗✗       1  NAPI_THROW_IF_FAILED(env, status, Buffer<T>());
1661    ✓✗        1  return Buffer(env, value, length, data);
1662                }
1663
1664                template <typename T>
1665                template <typename Finalizer>
1666               1inline Buffer<T> Buffer<T>::New(napi_env env,
1667                                               T* data,
1668                                               size_t length,
1669                                               Finalizer finalizeCallback) {
1670                  napi_value value;
1671                  details::FinalizeData<T, Finalizer>* finalizeData =
1672    ✓✗        1    new details::FinalizeData<T, Finalizer>({ finalizeCallback, nullptr });
1673                  napi_status status = napi_create_external_buffer(
1674                    env,
1675                    length * sizeof (T),
1676                    data,
1677                    details::FinalizeData<T, Finalizer>::Wrapper,
1678                    finalizeData,
1679    ✓✗        1    &value);
1680    ✗✓        1  if (status != napi_ok) {
1681                    delete finalizeData;
1682                    NAPI_THROW_IF_FAILED(env, status, Buffer());
1683                  }
1684    ✓✗        1  return Buffer(env, value, length, data);
1685                }
1686
1687                template <typename T>
1688                template <typename Finalizer, typename Hint>
1689               1inline Buffer<T> Buffer<T>::New(napi_env env,
1690                                               T* data,
1691                                               size_t length,
1692                                               Finalizer finalizeCallback,
1693                                               Hint* finalizeHint) {
1694                  napi_value value;
1695                  details::FinalizeData<T, Finalizer, Hint>* finalizeData =
1696    ✓✗        1    new details::FinalizeData<T, Finalizer, Hint>({ finalizeCallback, finalizeHint });
1697                  napi_status status = napi_create_external_buffer(
1698                    env,
1699                    length * sizeof (T),
1700                    data,
1701                    details::FinalizeData<T, Finalizer, Hint>::WrapperWithHint,
1702                    finalizeData,
1703    ✓✗        1    &value);
1704    ✗✓        1  if (status != napi_ok) {
1705                    delete finalizeData;
1706                    NAPI_THROW_IF_FAILED(env, status, Buffer());
1707                  }
1708    ✓✗        1  return Buffer(env, value, length, data);
1709                }
1710
1711                template <typename T>
1712               1inline Buffer<T> Buffer<T>::Copy(napi_env env, const T* data, size_t length) {
1713                  napi_value value;
1714                  napi_status status = napi_create_buffer_copy(
1715    ✓✗        1    env, length * sizeof (T), data, nullptr, &value);
1716 ✗✓✗✗       1  NAPI_THROW_IF_FAILED(env, status, Buffer<T>());
1717    ✓✗        1  return Buffer<T>(env, value);
1718                }
```

```
1719
1720                template <typename T>
1721                inline Buffer<T>::Buffer() : Uint8Array(), _length(0), _data(nullptr) {
1722                }
1723
1724                template <typename T>
1725               7inline Buffer<T>::Buffer(napi_env env, napi_value value)
1726               7  : Uint8Array(env, value), _length(0), _data(nullptr) {
1727               7}
1728
1729                template <typename T>
1730               4inline Buffer<T>::Buffer(napi_env env, napi_value value, size_t length, T* data)
1731               4  : Uint8Array(env, value), _length(length), _data(data) {
1732               4}
1733
1734                template <typename T>
1735              12inline size_t Buffer<T>::Length() const {
1736              12  EnsureInfo();
1737              12  return _length;
1738                }
1739
1740                template <typename T>
1741              12inline T* Buffer<T>::Data() const {
1742              12  EnsureInfo();
1743              12  return _data;
1744                }
1745
1746                template <typename T>
1747              24inline void Buffer<T>::EnsureInfo() const {
1748                  // The Buffer instance may have been constructed from a napi_value whose
1749                  // length/data are not yet known. Fetch and cache these values just once,
1750                  // since they can never change during the lifetime of the Buffer.
1751    ✓✓        24  if (_data == nullptr) {
1752                    size_t byteLength;
1753                    void* voidData;
1754    ✓✗         7    napi_status status = napi_get_buffer_info(_env, _value, &voidData, &byteLength);
1755 ✗✓✗✗         7    NAPI_THROW_IF_FAILED(_env, status);
1756               7    _length = byteLength / sizeof (T);
1757               7    _data = static_cast<T*>(voidData);
1758                  }
1759              24}
1760
1761                ////////////////////////////////////////////////////////////////////////////////
1762                // Error class
1763                ////////////////////////////////////////////////////////////////////////////////
1764
1765              31inline Error Error::New(napi_env env) {
1766                  napi_status status;
1767              31  napi_value error = nullptr;
1768
1769                  const napi_extended_error_info* info;
1770    ✓✗        31  status = napi_get_last_error_info(env, &info);
1771    ✗✓        31  NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_get_last_error_info");
1772
1773    ✓✗        31  if (status == napi_ok) {
1774    ✓✓        31    if (info->error_code == napi_pending_exception) {
1775    ✓✗         4      status = napi_get_and_clear_last_exception(env, &error);
1776    ✗✓         4      NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_get_and_clear_last_exception");
1777                    }
1778                    else {
1779              27      const char* error_message = info->error_message != nullptr ?
1780    ✓✗        27        info->error_message : "Error in native callback";
1781
1782                      bool isExceptionPending;
1783    ✓✗        27      status = napi_is_exception_pending(env, &isExceptionPending);
1784    ✗✓        27      NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_is_exception_pending");
1785
1786    ✓✓        27      if (isExceptionPending) {
1787    ✓✗        20        status = napi_get_and_clear_last_exception(env, &error);
1788    ✗✓        20        NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_get_and_clear_last_exception");
1789                      }
1790
1791                      napi_value message;
1792                      status = napi_create_string_utf8(
1793                        env,
1794                        error_message,
1795                        std::strlen(error_message),
```

```
1796 ✓✗      27        &message);
1797 ✗✓      27      NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_create_string_utf8");
1798
1799 ✓✗      27      if (status == napi_ok) {
1800 ✗✓      27        switch (info->error_code) {
1801                    case napi_object_expected:
1802                    case napi_string_expected:
1803                    case napi_boolean_expected:
1804                    case napi_number_expected:
1805                      status = napi_create_type_error(env, nullptr, message, &error);
1806                      break;
1807                    default:
1808 ✓✗      27           status = napi_create_error(env, nullptr,  message, &error);
1809          27           break;
1810                    }
1811 ✗✓      27        NAPI_FATAL_IF_FAILED(status, "Error::New", "napi_create_error");
1812                  }
1813                }
1814              }
1815
1816 ✓✗      31  return Error(env, error);
1817            }
1818
1819            inline Error Error::New(napi_env env, const char* message) {
1820              return Error::New<Error>(env, message, std::strlen(message), napi_create_error);
1821            }
1822
1823           4inline Error Error::New(napi_env env, const std::string& message) {
1824           4  return Error::New<Error>(env, message.c_str(), message.size(), napi_create_error);
1825            }
1826
1827            inline NAPI_NO_RETURN void Error::Fatal(const char* location, const char* message) {
1828              napi_fatal_error(location, NAPI_AUTO_LENGTH, message, NAPI_AUTO_LENGTH);
1829            }
1830
1831            inline Error::Error() : ObjectReference() {
1832            }
1833
1834 ✓✗      56inline Error::Error(napi_env env, napi_value value) : ObjectReference(env, nullptr) {
1835 ✓✗      56  if (value != nullptr) {
1836 ✓✗      56    napi_status status = napi_create_reference(env, value, 1, &_ref);
1837
1838                // Avoid infinite recursion in the failure case.
1839                // Don't try to construct & throw another Error instance.
1840 ✗✓      56    NAPI_FATAL_IF_FAILED(status, "Error::Error", "napi_create_reference");
1841              }
1842          56}
1843
1844            inline Error::Error(Error&& other) : ObjectReference(std::move(other)) {
1845            }
1846
1847            inline Error& Error::operator =(Error&& other) {
1848              static_cast<Reference<Object>*>(this)->operator=(std::move(other));
1849              return *this;
1850            }
1851
1852            inline Error::Error(const Error& other) : ObjectReference(other) {
1853            }
1854
1855            inline Error& Error::operator =(Error& other) {
1856              Reset();
1857
1858              _env = other.Env();
1859              HandleScope scope(_env);
1860
1861              napi_value value = other.Value();
1862              if (value != nullptr) {
1863                napi_status status = napi_create_reference(_env, value, 1, &_ref);
1864                NAPI_THROW_IF_FAILED(_env, status, *this);
1865              }
1866
1867              return *this;
1868            }
1869
1870           1inline const std::string& Error::Message() const NAPI_NOEXCEPT {
1871 ✓✗✓✗     1  if (_message.size() == 0 && _env != nullptr) {
      ✓✗
```

```
1872                #ifdef NAPI_CPP_EXCEPTIONS
1873                    try {
1874 ✓✗✓✗ ✓✗✓✗  1          _message = Get("message").As<String>();
1875                    }
1876                    catch (...) {
1877                      // Catch all errors here, to include e.g. a std::bad_alloc from
1878                      // the std::string::operator=, because this method may not throw.
1879                    }
1880                #else // NAPI_CPP_EXCEPTIONS
1881                    _message = Get("message").As<String>();
1882                #endif // NAPI_CPP_EXCEPTIONS
1883                  }
1884                1 return _message;
1885                 }
1886
1887               53inline void Error::ThrowAsJavaScriptException() const {
1888 ✓✗✓✗          53  HandleScope scope(_env);
1889 ✓✗✓✗          53  if (!IsEmpty()) {
1890 ✓✗✓✗ ✓✗       53    napi_status status = napi_throw(_env, Value());
1891 ✗✓✗✗          53    NAPI_THROW_IF_FAILED(_env, status);
1892               53  }
1893               53}
1894
1895                 #ifdef NAPI_CPP_EXCEPTIONS
1896
1897                 inline const char* Error::what() const NAPI_NOEXCEPT {
1898                   return Message().c_str();
1899                 }
1900
1901                 #endif // NAPI_CPP_EXCEPTIONS
1902
1903                 template <typename TError>
1904               25inline TError Error::New(napi_env env,
1905                                         const char* message,
1906                                         size_t length,
1907                                         create_error_fn create_error) {
1908                   napi_value str;
1909 ✓✗✓✗ ✗✗       25  napi_status status = napi_create_string_utf8(env, message, length, &str);
1910 ✗✓✗✗ ✗✓✗✗ ✗✗✗✗ 25  NAPI_THROW_IF_FAILED(env, status, TError());
1911
1912                   napi_value error;
1913 ✓✗✓✗ ✗✗       25  status = create_error(env, nullptr, str, &error);
1914 ✗✓✗✗ ✗✓✗✗ ✗✗✗✗ 25  NAPI_THROW_IF_FAILED(env, status, TError());
1915
1916 ✓✗✓✗ ✗✗       25  return TError(env, error);
1917                 }
1918
1919                 inline TypeError TypeError::New(napi_env env, const char* message) {
1920                   return Error::New<TypeError>(env, message, std::strlen(message), napi_create_type_error);
1921                 }
1922
1923                1inline TypeError TypeError::New(napi_env env, const std::string& message) {
1924                1  return Error::New<TypeError>(env, message.c_str(), message.size(), napi_create_type_error);
1925                 }
1926
1927                 inline TypeError::TypeError() : Error() {
1928                 }
1929
1930                1inline TypeError::TypeError(napi_env env, napi_value value) : Error(env, value) {
1931                1}
1932
1933               19inline RangeError RangeError::New(napi_env env, const char* message) {
1934               19  return Error::New<RangeError>(env, message, std::strlen(message), napi_create_range_error);
1935                 }
1936
1937                1inline RangeError RangeError::New(napi_env env, const std::string& message) {
1938                1  return Error::New<RangeError>(env, message.c_str(), message.size(), napi_create_range_error);
```

```
1939                }
1940
1941                inline RangeError::RangeError() : Error() {
1942                }
1943
1944              20inline RangeError::RangeError(napi_env env, napi_value value) : Error(env, value) {
1945              20}
1946
1947                ////////////////////////////////////////////////////////////////////////////////
1948                // Reference<T> class
1949                ////////////////////////////////////////////////////////////////////////////////
1950
1951                template <typename T>
1952              71inline Reference<T> Reference<T>::New(const T& value, uint32_t initialRefcount) {
1953 ✓✗✓✗ ✓✗✓✗    71  napi_env env = value.Env();
1954 ✓✗✓✗         71  napi_value val = value;
1955
1956 ✗✓✗✓         71  if (val == nullptr) {
1957                    return Reference<T>(env, nullptr);
1958                  }
1959
1960                  napi_ref ref;
1961 ✓✗✓✗ ✓✗✓✗    71  napi_status status = napi_create_reference(env, value, initialRefcount, &ref);
1962 ✗✓✗✗ ✗✓✗✗    71  NAPI_THROW_IF_FAILED(env, status, Reference<T>());
1963
1964 ✓✗✓✗         71  return Reference<T>(env, ref);
1965                }
1966
1967
1968                template <typename T>
1969              14inline Reference<T>::Reference() : _env(nullptr), _ref(nullptr), _suppressDestruct(false) {
1970              14}
1971
1972                template <typename T>
1973             135inline Reference<T>::Reference(napi_env env, napi_ref ref)
1974             135  : _env(env), _ref(ref), _suppressDestruct(false) {
1975             135}
1976
1977                template <typename T>
1978             199inline Reference<T>::~Reference() {
1979 ✓✓✓✓        199  if (_ref != nullptr) {
1980 ✓✗✓✓         78    if (!_suppressDestruct) {
1981              72      napi_delete_reference(_env, _ref);
1982                    }
1983
1984              78    _ref = nullptr;
1985                  }
1986             199}
1987
1988                template <typename T>
1989              50inline Reference<T>::Reference(Reference<T>&& other)
1990              50  : _env(other._env), _ref(other._ref), _suppressDestruct(other._suppressDestruct) {
1991              50  other._env = nullptr;
1992              50  other._ref = nullptr;
1993              50  other._suppressDestruct = false;
1994              50}
1995
1996                template <typename T>
1997              71inline Reference<T>& Reference<T>::operator =(Reference<T>&& other) {
1998              71  Reset();
1999              71  _env = other._env;
2000              71  _ref = other._ref;
2001              71  _suppressDestruct = other._suppressDestruct;
2002              71  other._env = nullptr;
2003              71  other._ref = nullptr;
2004              71  other._suppressDestruct = false;
2005              71  return *this;
2006                }
2007
2008                template <typename T>
2009                inline Reference<T>::Reference(const Reference<T>& other)
2010                  : _env(other._env), _ref(nullptr), _suppressDestruct(false) {
2011                  HandleScope scope(_env);
2012
2013                  napi_value value = other.Value();
```

```
2014                 if (value != nullptr) {
2015                   // Copying is a limited scenario (currently only used for Error object) and always creates a
2016                   // strong reference to the given value even if the incoming reference is weak.
2017                   napi_status status = napi_create_reference(_env, value, 1, &_ref);
2018                   NAPI_FATAL_IF_FAILED(status, "Reference<T>::Reference", "napi_create_reference");
2019                 }
2020               }
2021
2022               template <typename T>
2023               inline Reference<T>::operator napi_ref() const {
2024                 return _ref;
2025               }
2026
2027               template <typename T>
2028               inline bool Reference<T>::operator ==(const Reference<T> &other) const {
2029                 HandleScope scope(_env);
2030                 return this->Value().StrictEquals(other.Value());
2031               }
2032
2033               template <typename T>
2034               inline bool Reference<T>::operator !=(const Reference<T> &other) const {
2035                 return !this->operator ==(other);
2036               }
2037
2038               template <typename T>
2039               inline Napi::Env Reference<T>::Env() const {
2040                 return Napi::Env(_env);
2041               }
2042
2043               template <typename T>
2044            66 inline bool Reference<T>::IsEmpty() const {
2045            66   return _ref == nullptr;
2046                 }
2047
2048               template <typename T>
2049           161 inline T Reference<T>::Value() const {
2050 ✗✓✗✓      161   if (_ref == nullptr) {
2051                     return T(_env, nullptr);
2052                   }
2053
2054                   napi_value value;
2055 ✓✗✓✗      161   napi_status status = napi_get_reference_value(_env, _ref, &value);
2056 ✗✓✗✗      161   NAPI_THROW_IF_FAILED(_env, status, T());
    ✗✓✗✗
2057 ✓✗✓✗      161   return T(_env, value);
2058                 }
2059
2060               template <typename T>
2061             4 inline uint32_t Reference<T>::Ref() {
2062                 uint32_t result;
2063     ✓✗        4   napi_status status = napi_reference_ref(_env, _ref, &result);
2064 ✗✓✗✗        4   NAPI_THROW_IF_FAILED(_env, status, 1);
2065             4   return result;
2066               }
2067
2068               template <typename T>
2069            10 inline uint32_t Reference<T>::Unref() {
2070                 uint32_t result;
2071     ✓✗       10   napi_status status = napi_reference_unref(_env, _ref, &result);
2072 ✓✓✓✗       10   NAPI_THROW_IF_FAILED(_env, status, 1);
2073              7   return result;
2074               }
2075
2076               template <typename T>
2077            71 inline void Reference<T>::Reset() {
2078     ✓✓       71   if (_ref != nullptr) {
2079            57     napi_status status = napi_delete_reference(_env, _ref);
2080 ✗✓✗✗       57     NAPI_THROW_IF_FAILED(_env, status);
2081            57     _ref = nullptr;
2082                 }
2083            71 }
2084
2085               template <typename T>
2086               inline void Reference<T>::Reset(const T& value, uint32_t refcount) {
2087                 Reset();
2088                 _env = value.Env();
2089
2090                 napi_value val = value;
```

```
2091                 if (val != nullptr) {
2092                   napi_status status = napi_create_reference(_env, value, refcount, &_ref);
2093                   NAPI_THROW_IF_FAILED(_env, status);
2094                 }
2095               }
2096
2097               template <typename T>
2098             63inline void Reference<T>::SuppressDestruct() {
2099             63  _suppressDestruct = true;
2100             63}
2101
2102               template <typename T>
2103               inline Reference<T> Weak(T value) {
2104                 return Reference<T>::New(value, 0);
2105               }
2106
2107             21inline ObjectReference Weak(Object value) {
2108    ✓✗       21  return Reference<Object>::New(value, 0);
2109               }
2110
2111               inline FunctionReference Weak(Function value) {
2112                 return Reference<Function>::New(value, 0);
2113               }
2114
2115               template <typename T>
2116               inline Reference<T> Persistent(T value) {
2117                 return Reference<T>::New(value, 1);
2118               }
2119
2120             23inline ObjectReference Persistent(Object value) {
2121    ✓✗       23  return Reference<Object>::New(value, 1);
2122               }
2123
2124              6inline FunctionReference Persistent(Function value) {
2125    ✓✗        6  return Reference<Function>::New(value, 1);
2126               }
2127
2128               ////////////////////////////////////////////////////////////////////////////////
2129               // ObjectReference class
2130               ////////////////////////////////////////////////////////////////////////////////
2131
2132              6inline ObjectReference::ObjectReference(): Reference<Object>() {
2133              6}
2134
2135             56inline ObjectReference::ObjectReference(napi_env env, napi_ref ref): Reference<Object>(env, ref) {
2136             56}
2137
2138             44inline ObjectReference::ObjectReference(Reference<Object>&& other)
2139             44  : Reference<Object>(std::move(other)) {
2140             44}
2141
2142             21inline ObjectReference& ObjectReference::operator =(Reference<Object>&& other) {
2143             21  static_cast<Reference<Object>*>(this)->operator=(std::move(other));
2144             21  return *this;
2145               }
2146
2147               inline ObjectReference::ObjectReference(ObjectReference&& other)
2148                 : Reference<Object>(std::move(other)) {
2149               }
2150
2151             42inline ObjectReference& ObjectReference::operator =(ObjectReference&& other) {
2152             42  static_cast<Reference<Object>*>(this)->operator=(std::move(other));
2153             42  return *this;
2154               }
2155
2156               inline ObjectReference::ObjectReference(const ObjectReference& other)
2157                 : Reference<Object>(other) {
2158               }
2159
2160              1inline Napi::Value ObjectReference::Get(const char* utf8name) const {
2161 ✓✗✓✗          1  EscapableHandleScope scope(_env);
     ✓✗✓✗
2162 ✓✗✓✗          1  return scope.Escape(Value().Get(utf8name));
2163               }
2164
2165              3inline Napi::Value ObjectReference::Get(const std::string& utf8name) const {
2166 ✓✗✓✗          3  EscapableHandleScope scope(_env);
     ✓✗✓✗
2167 ✓✗✓✗          3  return scope.Escape(Value().Get(utf8name));
```

```
2168                }
2169
2170                inline void ObjectReference::Set(const char* utf8name, napi_value value) {
2171                  HandleScope scope(_env);
2172                  Value().Set(utf8name, value);
2173                }
2174
2175               4inline void ObjectReference::Set(const char* utf8name, Napi::Value value) {
2176 ✓✗✓✗          4  HandleScope scope(_env);
2177 ✓✗✓✗          4  Value().Set(utf8name, value);
2178               4}
2179
2180                inline void ObjectReference::Set(const char* utf8name, const char* utf8value) {
2181                  HandleScope scope(_env);
2182                  Value().Set(utf8name, utf8value);
2183                }
2184
2185                inline void ObjectReference::Set(const char* utf8name, bool boolValue) {
2186                  HandleScope scope(_env);
2187                  Value().Set(utf8name, boolValue);
2188                }
2189
2190                inline void ObjectReference::Set(const char* utf8name, double numberValue) {
2191                  HandleScope scope(_env);
2192                  Value().Set(utf8name, numberValue);
2193                }
2194
2195                inline void ObjectReference::Set(const std::string& utf8name, napi_value value) {
2196                  HandleScope scope(_env);
2197                  Value().Set(utf8name, value);
2198                }
2199
2200              27inline void ObjectReference::Set(const std::string& utf8name, Napi::Value value) {
2201 ✓✗✓✗         27  HandleScope scope(_env);
2202 ✓✗✓✗         27  Value().Set(utf8name, value);
2203              27}
2204
2205                inline void ObjectReference::Set(const std::string& utf8name, std::string& utf8value) {
2206                  HandleScope scope(_env);
2207                  Value().Set(utf8name, utf8value);
2208                }
2209
2210                inline void ObjectReference::Set(const std::string& utf8name, bool boolValue) {
2211                  HandleScope scope(_env);
2212                  Value().Set(utf8name, boolValue);
2213                }
2214
2215                inline void ObjectReference::Set(const std::string& utf8name, double numberValue) {
2216                  HandleScope scope(_env);
2217                  Value().Set(utf8name, numberValue);
2218                }
2219
2220              18inline Napi::Value ObjectReference::Get(uint32_t index) const {
2221 ✓✗✓✗         18  EscapableHandleScope scope(_env);
2222 ✓✗✓✗ ✓✗✓✗    18  return scope.Escape(Value().Get(index));
2223                }
2224
2225                inline void ObjectReference::Set(uint32_t index, napi_value value) {
2226                  HandleScope scope(_env);
2227                  Value().Set(index, value);
2228                }
2229
2230              27inline void ObjectReference::Set(uint32_t index, Napi::Value value) {
2231 ✓✗✓✗         27  HandleScope scope(_env);
2232 ✓✗✓✗         27  Value().Set(index, value);
2233              27}
2234
2235                inline void ObjectReference::Set(uint32_t index, const char* utf8value) {
2236                  HandleScope scope(_env);
2237                  Value().Set(index, utf8value);
2238                }
2239
2240                inline void ObjectReference::Set(uint32_t index, const std::string& utf8value) {
2241                  HandleScope scope(_env);
2242                  Value().Set(index, utf8value);
2243                }
2244
2245                inline void ObjectReference::Set(uint32_t index, bool boolValue) {
```

```
2246                HandleScope scope(_env);
2247                Value().Set(index, boolValue);
2248              }
2249
2250              inline void ObjectReference::Set(uint32_t index, double numberValue) {
2251                HandleScope scope(_env);
2252                Value().Set(index, numberValue);
2253              }
2254
2255              ////////////////////////////////////////////////////////////////////////////
2256              // FunctionReference class
2257              ////////////////////////////////////////////////////////////////////////////
2258
2259              inline FunctionReference::FunctionReference(): Reference<Function>() {
2260              }
2261
2262              inline FunctionReference::FunctionReference(napi_env env, napi_ref ref)
2263                : Reference<Function>(env, ref) {
2264              }
2265
2266            6 inline FunctionReference::FunctionReference(Reference<Function>&& other)
2267            6   : Reference<Function>(std::move(other)) {
2268            6 }
2269
2270              inline FunctionReference& FunctionReference::operator =(Reference<Function>&& other) {
2271                static_cast<Reference<Function>*>(this)->operator=(std::move(other));
2272                return *this;
2273              }
2274
2275              inline FunctionReference::FunctionReference(FunctionReference&& other)
2276                : Reference<Function>(std::move(other)) {
2277              }
2278
2279              inline FunctionReference& FunctionReference::operator =(FunctionReference&& other) {
2280                static_cast<Reference<Function>*>(this)->operator=(std::move(other));
2281                return *this;
2282              }
2283
2284              inline Napi::Value FunctionReference::operator ()(
2285                  const std::initializer_list<napi_value>& args) const {
2286                EscapableHandleScope scope(_env);
2287                return scope.Escape(Value()(args));
2288              }
2289
2290              inline Napi::Value FunctionReference::Call(const std::initializer_list<napi_value>& args) const {
2291                EscapableHandleScope scope(_env);
2292                Napi::Value result = Value().Call(args);
2293                if (scope.Env().IsExceptionPending()) {
2294                  return Value();
2295                }
2296                return scope.Escape(result);
2297              }
2298
2299              inline Napi::Value FunctionReference::Call(const std::vector<napi_value>& args) const {
2300                EscapableHandleScope scope(_env);
2301                Napi::Value result = Value().Call(args);
2302                if (scope.Env().IsExceptionPending()) {
2303                  return Value();
2304                }
2305                return scope.Escape(result);
2306              }
2307
2308              inline Napi::Value FunctionReference::Call(
2309                  napi_value recv, const std::initializer_list<napi_value>& args) const {
2310                EscapableHandleScope scope(_env);
2311                Napi::Value result = Value().Call(recv, args);
2312                if (scope.Env().IsExceptionPending()) {
2313                  return Value();
2314                }
2315                return scope.Escape(result);
2316              }
2317
2318              inline Napi::Value FunctionReference::Call(
2319                  napi_value recv, const std::vector<napi_value>& args) const {
2320                EscapableHandleScope scope(_env);
2321                Napi::Value result = Value().Call(recv, args);
2322                if (scope.Env().IsExceptionPending()) {
2323                  return Value();
2324                }
2325                return scope.Escape(result);
2326              }
2327
```

```
2328                2 inline Napi::Value FunctionReference::MakeCallback(
2329                      napi_value recv, const std::initializer_list<napi_value>& args) const {
2330 ✓✗✓✗           2   EscapableHandleScope scope(_env);
2331 ✓✗✓✗           2   Napi::Value result = Value().MakeCallback(recv, args);
2332 ✓✗✓✗           2   if (scope.Env().IsExceptionPending()) {
    ✗✓
2333                      return Value();
2334                    }
2335 ✓✗✓✗           2   return scope.Escape(result);
2336                  }
2337
2338                  inline Napi::Value FunctionReference::MakeCallback(
2339                      napi_value recv, const std::vector<napi_value>& args) const {
2340                    EscapableHandleScope scope(_env);
2341                    Napi::Value result = Value().MakeCallback(recv, args);
2342                    if (scope.Env().IsExceptionPending()) {
2343                      return Value();
2344                    }
2345                    return scope.Escape(result);
2346                  }
2347
2348                4 inline Object FunctionReference::New(const std::initializer_list<napi_value>& args) const {
2349 ✓✗✓✗           4   EscapableHandleScope scope(_env);
    ✓✗✓✗
2350 ✓✗✓✗           4   return scope.Escape(Value().New(args)).As<Object>();
       ✓✗
2351                  }
2352
2353                  inline Object FunctionReference::New(const std::vector<napi_value>& args) const {
2354                    EscapableHandleScope scope(_env);
2355                    return scope.Escape(Value().New(args)).As<Object>();
2356                  }
2357
2358                  ////////////////////////////////////////////////////////////////////////////////
2359                  // CallbackInfo class
2360                  ////////////////////////////////////////////////////////////////////////////////
2361
2362           1921 inline CallbackInfo::CallbackInfo(napi_env env, napi_callback_info info)
2363           1921     : _env(env), _info(info), _this(nullptr), _dynamicArgs(nullptr), _data(nullptr) {
2364           1921   _argc = _staticArgCount;
2365           1921   _argv = _staticArgs;
2366           1921   napi_status status = napi_get_cb_info(env, info, &_argc, _argv, &_this, &_data);
2367 ✗✓✗✗      1921   NAPI_THROW_IF_FAILED(_env, status);
2368
2369    ✓✓     1921   if (_argc > _staticArgCount) {
2370                    // Use either a fixed-size array (on the stack) or a dynamically-allocated
2371                    // array (on the heap) depending on the number of args.
2372    ✓✗        1     _dynamicArgs = new napi_value[_argc];
2373                1     _argv = _dynamicArgs;
2374
2375                1     status = napi_get_cb_info(env, info, &_argc, _argv, nullptr, nullptr);
2376 ✗✓✗✗           1     NAPI_THROW_IF_FAILED(_env, status);
2377                    }
2378           1921 }
2379
2380           1921 inline CallbackInfo::~CallbackInfo() {
2381    ✓✓     1921   if (_dynamicArgs != nullptr) {
2382    ✓✗        1     delete[] _dynamicArgs;
2383                  }
2384           1921 }
2385
2386                2 inline Value CallbackInfo::NewTarget() const {
2387                    napi_value newTarget;
2388    ✓✗        2   napi_status status = napi_get_new_target(_env, _info, &newTarget);
2389 ✗✓✗✗           2   NAPI_THROW_IF_FAILED(_env, status, Value());
2390    ✓✗        2   return Value(_env, newTarget);
2391                  }
2392
2393                2 inline bool CallbackInfo::IsConstructCall() const {
2394    ✓✗        2   return !NewTarget().IsEmpty();
2395                  }
2396
2397        2001762 inline Napi::Env CallbackInfo::Env() const {
2398        2001762   return Napi::Env(_env);
2399                  }
2400
```

```
2401                inline size_t CallbackInfo::Length() const {
2402                  return _argc;
2403                }
2404
2405            2465inline const Value CallbackInfo::operator [](size_t index) const {
      ✓✓✓✗
2406  ✓✗✓✗      2465  return index < _argc ? Value(_env, _argv[index]) : Env().Undefined();
      ✓✓✗✗
2407                }
2408
2409              64inline Value CallbackInfo::This() const {
2410     ✗✓       64  if (_this == nullptr) {
2411                    return Env().Undefined();
2412                  }
2413              64  return Object(_env, _this);
2414                }
2415
2416            1915inline void* CallbackInfo::Data() const {
2417            1915  return _data;
2418                }
2419
2420            1895inline void CallbackInfo::SetData(void* data) {
2421            1895  _data = data;
2422            1895}
2423
2424                ////////////////////////////////////////////////////////////////////////////////
2425                // PropertyDescriptor class
2426                ////////////////////////////////////////////////////////////////////////////////
2427
2428                template <typename Getter>
2429                inline PropertyDescriptor
2430               2PropertyDescriptor::Accessor(const char* utf8name,
2431                                             Getter getter,
2432                                             napi_property_attributes attributes,
2433                                             void* data) {
2434                  typedef details::CallbackData<Getter, Napi::Value> CbData;
2435                  // TODO: Delete when the function is destroyed
2436               2  auto callbackData = new CbData({ getter, nullptr });
2437
2438                  return PropertyDescriptor({
2439                    utf8name,
2440                    nullptr,
2441                    nullptr,
2442                    CbData::Wrapper,
2443                    nullptr,
2444                    nullptr,
2445                    attributes,
2446                    callbackData
2447               2  });
2448                }
2449
2450                template <typename Getter>
2451               1inline PropertyDescriptor PropertyDescriptor::Accessor(const std::string& utf8name,
2452                                                                       Getter getter,
2453                                                                       napi_property_attributes attributes,
2454                                                                       void* data) {
2455               1  return Accessor(utf8name.c_str(), getter, attributes, data);
2456                }
2457
2458                template <typename Getter>
2459               1inline PropertyDescriptor PropertyDescriptor::Accessor(napi_value name,
2460                                                                       Getter getter,
2461                                                                       napi_property_attributes attributes,
2462                                                                       void* data) {
2463                  typedef details::CallbackData<Getter, Napi::Value> CbData;
2464                  // TODO: Delete when the function is destroyed
2465               1  auto callbackData = new CbData({ getter, nullptr });
2466
2467                  return PropertyDescriptor({
2468                    nullptr,
2469                    name,
2470                    nullptr,
2471                    CbData::Wrapper,
2472                    nullptr,
2473                    nullptr,
2474                    attributes,
2475                    callbackData
2476               1  });
2477                }
2478
2479                template <typename Getter>
```

```
2480              1inline PropertyDescriptor PropertyDescriptor::Accessor(Name name,
2481                                                                      Getter getter,
2482                                                                      napi_property_attributes attributes,
2483                                                                      void* data) {
2484              1  napi_value nameValue = name;
2485              1  return PropertyDescriptor::Accessor(nameValue, getter, attributes, data);
2486               }
2487
2488               template <typename Getter, typename Setter>
2489              2inline PropertyDescriptor PropertyDescriptor::Accessor(const char* utf8name,
2490                                                                      Getter getter,
2491                                                                      Setter setter,
2492                                                                      napi_property_attributes attributes,
2493                                                                      void* data) {
2494                 typedef details::AccessorCallbackData<Getter, Setter> CbData;
2495                 // TODO: Delete when the function is destroyed
2496              2  auto callbackData = new CbData({ getter, setter });
2497
2498                 return PropertyDescriptor({
2499                   utf8name,
2500                   nullptr,
2501                   nullptr,
2502                   CbData::GetterWrapper,
2503                   CbData::SetterWrapper,
2504                   nullptr,
2505                   attributes,
2506                   callbackData
2507              2  });
2508               }
2509
2510               template <typename Getter, typename Setter>
2511              1inline PropertyDescriptor PropertyDescriptor::Accessor(const std::string& utf8name,
2512                                                                      Getter getter,
2513                                                                      Setter setter,
2514                                                                      napi_property_attributes attributes,
2515                                                                      void* data) {
2516              1  return Accessor(utf8name.c_str(), getter, setter, attributes, data);
2517               }
2518
2519               template <typename Getter, typename Setter>
2520              1inline PropertyDescriptor PropertyDescriptor::Accessor(napi_value name,
2521                                                                      Getter getter,
2522                                                                      Setter setter,
2523                                                                      napi_property_attributes attributes,
2524                                                                      void* data) {
2525                 typedef details::AccessorCallbackData<Getter, Setter> CbData;
2526                 // TODO: Delete when the function is destroyed
2527              1  auto callbackData = new CbData({ getter, setter });
2528
2529                 return PropertyDescriptor({
2530                   nullptr,
2531                   name,
2532                   nullptr,
2533                   CbData::GetterWrapper,
2534                   CbData::SetterWrapper,
2535                   nullptr,
2536                   attributes,
2537                   callbackData
2538              1  });
2539               }
2540
2541               template <typename Getter, typename Setter>
2542              1inline PropertyDescriptor PropertyDescriptor::Accessor(Name name,
2543                                                                      Getter getter,
2544                                                                      Setter setter,
2545                                                                      napi_property_attributes attributes,
2546                                                                      void* data) {
2547              1  napi_value nameValue = name;
2548              1  return PropertyDescriptor::Accessor(nameValue, getter, setter, attributes, data);
2549               }
2550
2551               template <typename Callable>
2552              2inline PropertyDescriptor PropertyDescriptor::Function(const char* utf8name,
2553                                                                      Callable cb,
2554                                                                      napi_property_attributes attributes,
2555                                                                      void* data) {
2556                 typedef decltype(cb(CallbackInfo(nullptr, nullptr))) ReturnType;
2557                 typedef details::CallbackData<Callable, ReturnType> CbData;
2558                 // TODO: Delete when the function is destroyed
2559              2  auto callbackData = new CbData({ cb, nullptr });
2560
2561                 return PropertyDescriptor({
```

```
2562                   utf8name,
2563                   nullptr,
2564                   CbData::Wrapper,
2565                   nullptr,
2566                   nullptr,
2567                   nullptr,
2568                   attributes,
2569                   callbackData
2570               2  });
2571                }
2572
2573                template <typename Callable>
2574               1inline PropertyDescriptor PropertyDescriptor::Function(const std::string& utf8name,
2575                                                                       Callable cb,
2576                                                                       napi_property_attributes attributes,
2577                                                                       void* data) {
2578               1  return Function(utf8name.c_str(), cb, attributes, data);
2579                }
2580
2581                template <typename Callable>
2582               1inline PropertyDescriptor PropertyDescriptor::Function(napi_value name,
2583                                                                       Callable cb,
2584                                                                       napi_property_attributes attributes,
2585                                                                       void* data) {
2586                  typedef decltype(cb(CallbackInfo(nullptr, nullptr))) ReturnType;
2587                  typedef details::CallbackData<Callable, ReturnType> CbData;
2588                  // TODO: Delete when the function is destroyed
2589               1  auto callbackData = new CbData({ cb, nullptr });
2590
2591                  return PropertyDescriptor({
2592                    nullptr,
2593                    name,
2594                    CbData::Wrapper,
2595                    nullptr,
2596                    nullptr,
2597                    nullptr,
2598                    attributes,
2599                    callbackData
2600               1  });
2601                }
2602
2603                template <typename Callable>
2604               1inline PropertyDescriptor PropertyDescriptor::Function(Name name,
2605                                                                       Callable cb,
2606                                                                       napi_property_attributes attributes,
2607                                                                       void* data) {
2608               1  napi_value nameValue = name;
2609               1  return PropertyDescriptor::Function(nameValue, cb, attributes, data);
2610                }
2611
2612               8inline PropertyDescriptor PropertyDescriptor::Value(const char* utf8name,
2613                                                                   napi_value value,
2614                                                                   napi_property_attributes attributes) {
2615                  return PropertyDescriptor({
2616                    utf8name, nullptr, nullptr, nullptr, nullptr, value, attributes, nullptr
2617               8  });
2618                }
2619
2620               4inline PropertyDescriptor PropertyDescriptor::Value(const std::string& utf8name,
2621                                                                   napi_value value,
2622                                                                   napi_property_attributes attributes) {
2623               4  return Value(utf8name.c_str(), value, attributes);
2624                }
2625
2626               5inline PropertyDescriptor PropertyDescriptor::Value(napi_value name,
2627                                                                   napi_value value,
2628                                                                   napi_property_attributes attributes) {
2629                  return PropertyDescriptor({
2630                    nullptr, name, nullptr, nullptr, nullptr, value, attributes, nullptr
2631               5  });
2632                }
2633
2634               5inline PropertyDescriptor PropertyDescriptor::Value(Name name,
2635                                                                   Napi::Value value,
2636                                                                   napi_property_attributes attributes) {
2637               5  napi_value nameValue = name;
2638               5  napi_value valueValue = value;
2639               5  return PropertyDescriptor::Value(nameValue, valueValue, attributes);
2640                }
2641
2642              22inline PropertyDescriptor::PropertyDescriptor(napi_property_descriptor desc)
2643              22  : _desc(desc) {
```

```
2644              22 }
2645
2646                 inline PropertyDescriptor::operator napi_property_descriptor&() {
2647                   return _desc;
2648                 }
2649
2650                 inline PropertyDescriptor::operator const napi_property_descriptor&() const {
2651                   return _desc;
2652                 }
2653
2654                 ////////////////////////////////////////////////////////////////////////////////
2655                 // ObjectWrap<T> class
2656                 ////////////////////////////////////////////////////////////////////////////////
2657
2658                 template <typename T>
2659                8 inline ObjectWrap<T>::ObjectWrap(const Napi::CallbackInfo& callbackInfo) {
2660 ✓✗✓✗ ✓✗✓✗       8   napi_env env = callbackInfo.Env();
2661 ✓✗✓✗ ✓✗✓✗       8   napi_value wrapper = callbackInfo.This();
2662                    napi_status status;
2663                    napi_ref ref;
2664                8   T* instance = static_cast<T*>(this);
2665 ✓✗✓✗           8   status = napi_wrap(env, wrapper, instance, FinalizeCallback, nullptr, &ref);
2666 ✗✓✗✗ ✗✓✗✗       8   NAPI_THROW_IF_FAILED(env, status)
2667
2668                8   Reference<Object>* instanceRef = instance;
2669 ✓✗✓✗ ✓✗✓✗       8   *instanceRef = Reference<Object>(env, ref);
2670                8 }
2671
2672                  template<typename T>
2673               48 inline T* ObjectWrap<T>::Unwrap(Object wrapper) {
2674                    T* unwrapped;
     ✓✗✓✗
2675 ✓✗✓✗ ✓✗✓✗ ✓✗✓✗ ✗✓✗✗ ✗✗✗✗ ✗✓✗✗ ✗✗✗✗ 48   napi_status status = napi_unwrap(wrapper.Env(), wrapper, reinterpret_cast<void**>(&unwrapped));
2676 ✗✗✗✗ ✗✓✗✗ ✗✗✗✗ 48   NAPI_THROW_IF_FAILED(wrapper.Env(), status, nullptr);
2677               48   return unwrapped;
2678                  }
2679
2680                  template <typename T>
2681                5 inline Function ObjectWrap<T>::DefineClass(
2682                      Napi::Env env,
2683                      const char* utf8name,
2684                      const std::initializer_list<ClassPropertyDescriptor<T>>& properties,
2685                      void* data) {
2686                    napi_value value;
2687                    napi_status status = napi_define_class(
2688                      env, utf8name, NAPI_AUTO_LENGTH,
2689                      T::ConstructorCallbackWrapper, data, properties.size(),
2690 ✓✗✓✗ ✓✗✓✗ ✗✓✗✗ ✗✗✗✓ ✗✗✗✗       5     reinterpret_cast<const napi_property_descriptor*>(properties.begin()), &value);
2691                5   NAPI_THROW_IF_FAILED(env, status, Function());
2692
2693 ✓✗✓✗ ✓✗✓✗       5   return Function(env, value);
2694                  }
2695
2696                  template <typename T>
2697                  inline Function ObjectWrap<T>::DefineClass(
2698                      Napi::Env env,
2699                      const char* utf8name,
2700                      const std::vector<ClassPropertyDescriptor<T>>& properties,
2701                      void* data) {
2702                    napi_value value;
2703                    napi_status status = napi_define_class(
2704                      env, utf8name, NAPI_AUTO_LENGTH,
```

```
2705                  T::ConstructorCallbackWrapper, data, properties.size(),
2706                  reinterpret_cast<const napi_property_descriptor*>(properties.data()), &value);
2707                NAPI_THROW_IF_FAILED(env, status, Function());
2708
2709                return Function(env, value);
2710              }
2711
2712              template <typename T>
2713              inline ClassPropertyDescriptor<T> ObjectWrap<T>::StaticMethod(
2714                  const char* utf8name,
2715                  StaticVoidMethodCallback method,
2716                  napi_property_attributes attributes,
2717                  void* data) {
2718                // TODO: Delete when the class is destroyed
2719                StaticVoidMethodCallbackData* callbackData = new StaticVoidMethodCallbackData({ method, data });
2720
2721                napi_property_descriptor desc = napi_property_descriptor();
2722                desc.utf8name = utf8name;
2723                desc.method = T::StaticVoidMethodCallbackWrapper;
2724                desc.data = callbackData;
2725                desc.attributes = static_cast<napi_property_attributes>(attributes | napi_static);
2726                return desc;
2727              }
2728
2729              template <typename T>
2730              inline ClassPropertyDescriptor<T> ObjectWrap<T>::StaticMethod(
2731                  const char* utf8name,
2732                  StaticMethodCallback method,
2733                  napi_property_attributes attributes,
2734                  void* data) {
2735                // TODO: Delete when the class is destroyed
2736                StaticMethodCallbackData* callbackData = new StaticMethodCallbackData({ method, data });
2737
2738                napi_property_descriptor desc = napi_property_descriptor();
2739                desc.utf8name = utf8name;
2740                desc.method = T::StaticMethodCallbackWrapper;
2741                desc.data = callbackData;
2742                desc.attributes = static_cast<napi_property_attributes>(attributes | napi_static);
2743                return desc;
2744              }
2745
2746              template <typename T>
2747              inline ClassPropertyDescriptor<T> ObjectWrap<T>::StaticAccessor(
2748                  const char* utf8name,
2749                  StaticGetterCallback getter,
2750                  StaticSetterCallback setter,
2751                  napi_property_attributes attributes,
2752                  void* data) {
2753                // TODO: Delete when the class is destroyed
2754                StaticAccessorCallbackData* callbackData =
2755                  new StaticAccessorCallbackData({ getter, setter, data });
2756
2757                napi_property_descriptor desc = napi_property_descriptor();
2758                desc.utf8name = utf8name;
2759                desc.getter = getter != nullptr ? T::StaticGetterCallbackWrapper : nullptr;
2760                desc.setter = setter != nullptr ? T::StaticSetterCallbackWrapper : nullptr;
2761                desc.data = callbackData;
2762                desc.attributes = static_cast<napi_property_attributes>(attributes | napi_static);
2763                return desc;
2764              }
2765
2766              template <typename T>
2767            1 inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceMethod(
2768                  const char* utf8name,
2769                  InstanceVoidMethodCallback method,
2770                  napi_property_attributes attributes,
2771                  void* data) {
2772                // TODO: Delete when the class is destroyed
2773                InstanceVoidMethodCallbackData* callbackData =
2774    ✓✗        1     new InstanceVoidMethodCallbackData({ method, data});
2775
2776            1   napi_property_descriptor desc = napi_property_descriptor();
2777            1   desc.utf8name = utf8name;
2778            1   desc.method = T::InstanceVoidMethodCallbackWrapper;
2779            1   desc.data = callbackData;
2780            1   desc.attributes = attributes;
2781    ✓✗      1   return desc;
2782              }
2783
2784              template <typename T>
2785            5 inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceMethod(
```

```
2786                   const char* utf8name,
2787                   InstanceMethodCallback method,
2788                   napi_property_attributes attributes,
2789                   void* data) {
2790                 // TODO: Delete when the class is destroyed
2791 ✓✗✓✗       5   InstanceMethodCallbackData* callbackData = new InstanceMethodCallbackData({ method, data });
2792
2793              5   napi_property_descriptor desc = napi_property_descriptor();
2794              5   desc.utf8name = utf8name;
2795              5   desc.method = T::InstanceMethodCallbackWrapper;
2796              5   desc.data = callbackData;
2797              5   desc.attributes = attributes;
2798 ✓✗✓✗       5   return desc;
2799               }
2800
2801               template <typename T>
2802               inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceMethod(
2803                   Symbol name,
2804                   InstanceVoidMethodCallback method,
2805                   napi_property_attributes attributes,
2806                   void* data) {
2807                 // TODO: Delete when the class is destroyed
2808                 InstanceVoidMethodCallbackData* callbackData =
2809                   new InstanceVoidMethodCallbackData({ method, data});
2810
2811                 napi_property_descriptor desc = napi_property_descriptor();
2812                 desc.name = name;
2813                 desc.method = T::InstanceVoidMethodCallbackWrapper;
2814                 desc.data = callbackData;
2815                 desc.attributes = attributes;
2816                 return desc;
2817               }
2818
2819               template <typename T>
2820              1inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceMethod(
2821                   Symbol name,
2822                   InstanceMethodCallback method,
2823                   napi_property_attributes attributes,
2824                   void* data) {
2825                 // TODO: Delete when the class is destroyed
2826    ✓✗       1   InstanceMethodCallbackData* callbackData = new InstanceMethodCallbackData({ method, data });
2827
2828              1   napi_property_descriptor desc = napi_property_descriptor();
2829    ✓✗       1   desc.name = name;
2830              1   desc.method = T::InstanceMethodCallbackWrapper;
2831              1   desc.data = callbackData;
2832              1   desc.attributes = attributes;
2833    ✓✗       1   return desc;
2834               }
2835
2836               template <typename T>
2837              3inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceAccessor(
2838                   const char* utf8name,
2839                   InstanceGetterCallback getter,
2840                   InstanceSetterCallback setter,
2841                   napi_property_attributes attributes,
2842                   void* data) {
2843                 // TODO: Delete when the class is destroyed
2844                 InstanceAccessorCallbackData* callbackData =
2845    ✓✗       3     new InstanceAccessorCallbackData({ getter, setter, data });
2846
2847              3   napi_property_descriptor desc = napi_property_descriptor();
2848              3   desc.utf8name = utf8name;
2849    ✓✓       3   desc.getter = getter != nullptr ? T::InstanceGetterCallbackWrapper : nullptr;
2850    ✓✓       3   desc.setter = setter != nullptr ? T::InstanceSetterCallbackWrapper : nullptr;
2851              3   desc.data = callbackData;
2852              3   desc.attributes = attributes;
2853    ✓✗       3   return desc;
2854               }
2855
2856               template <typename T>
2857               inline ClassPropertyDescriptor<T> ObjectWrap<T>::StaticValue(const char* utf8name,
2858                   Napi::Value value, napi_property_attributes attributes) {
2859                 napi_property_descriptor desc = napi_property_descriptor();
2860                 desc.utf8name = utf8name;
2861                 desc.value = value;
2862                 desc.attributes = static_cast<napi_property_attributes>(attributes | napi_static);
2863                 return desc;
2864               }
```

```
2865
2866                template <typename T>
2867                inline ClassPropertyDescriptor<T> ObjectWrap<T>::InstanceValue(
2868                    const char* utf8name,
2869                    Napi::Value value,
2870                    napi_property_attributes attributes) {
2871                  napi_property_descriptor desc = napi_property_descriptor();
2872                  desc.utf8name = utf8name;
2873                  desc.value = value;
2874                  desc.attributes = attributes;
2875                  return desc;
2876                }
2877
2878                template <typename T>
2879               8inline napi_value ObjectWrap<T>::ConstructorCallbackWrapper(
2880                    napi_env env,
2881                    napi_callback_info info) {
2882                  napi_value new_target;
2883 ✓✗✓✗       8  napi_status status = napi_get_new_target(env, info, &new_target);
2884 ✗✓✗✓       8  if (status != napi_ok) return nullptr;
2885
2886                8  bool isConstructCall = (new_target != nullptr);
2887 ✗✓✗✓       8  if (!isConstructCall) {
2888                    napi_throw_type_error(env, nullptr, "Class constructors cannot be invoked without 'new'");
2889                    return nullptr;
2890                  }
2891
2892                  T* instance;
2893                8  napi_value wrapper = details::WrapCallback([&] {
2894 ✓✗✓✗       8    CallbackInfo callbackInfo(env, info);
     ✓✗✓✗
2895 ✓✗✓✗       8    instance = new T(callbackInfo);
2896 ✓✗✓✗      16    return callbackInfo.This();
2897 ✓✗✓✗      16  });
2898
2899                8  return wrapper;
2900               }
2901
2902               template <typename T>
2903               inline napi_value ObjectWrap<T>::StaticVoidMethodCallbackWrapper(
2904                   napi_env env,
2905                   napi_callback_info info) {
2906                 return details::WrapCallback([&] {
2907                   CallbackInfo callbackInfo(env, info);
2908                   StaticVoidMethodCallbackData* callbackData =
2909                     reinterpret_cast<StaticVoidMethodCallbackData*>(callbackInfo.Data());
2910                   callbackInfo.SetData(callbackData->data);
2911                   callbackData->callback(callbackInfo);
2912                   return nullptr;
2913                 });
2914               }
2915
2916               template <typename T>
2917               inline napi_value ObjectWrap<T>::StaticMethodCallbackWrapper(
2918                   napi_env env,
2919                   napi_callback_info info) {
2920                 return details::WrapCallback([&] {
2921                   CallbackInfo callbackInfo(env, info);
2922                   StaticMethodCallbackData* callbackData =
2923                     reinterpret_cast<StaticMethodCallbackData*>(callbackInfo.Data());
2924                   callbackInfo.SetData(callbackData->data);
2925                   return callbackData->callback(callbackInfo);
2926                 });
2927               }
2928
2929               template <typename T>
2930               inline napi_value ObjectWrap<T>::StaticGetterCallbackWrapper(
2931                   napi_env env,
2932                   napi_callback_info info) {
2933                 return details::WrapCallback([&] {
2934                   CallbackInfo callbackInfo(env, info);
2935                   StaticAccessorCallbackData* callbackData =
2936                     reinterpret_cast<StaticAccessorCallbackData*>(callbackInfo.Data());
2937                   callbackInfo.SetData(callbackData->data);
2938                   return callbackData->getterCallback(callbackInfo);
2939                 });
2940               }
2941
2942               template <typename T>
```

```
2943                inline napi_value ObjectWrap<T>::StaticSetterCallbackWrapper(
2944                    napi_env env,
2945                    napi_callback_info info) {
2946                  return details::WrapCallback([&] {
2947                    CallbackInfo callbackInfo(env, info);
2948                    StaticAccessorCallbackData* callbackData =
2949                      reinterpret_cast<StaticAccessorCallbackData*>(callbackInfo.Data());
2950                    callbackInfo.SetData(callbackData->data);
2951                    callbackData->setterCallback(callbackInfo, callbackInfo[0]);
2952                    return nullptr;
2953                  });
2954                }
2955
2956                template <typename T>
2957               8inline napi_value ObjectWrap<T>::InstanceVoidMethodCallbackWrapper(
2958                    napi_env env,
2959                    napi_callback_info info) {
2960               8  return details::WrapCallback([&] {
2961    ✓✗         8    CallbackInfo callbackInfo(env, info);
2962                      InstanceVoidMethodCallbackData* callbackData =
2963    ✓✗         8      reinterpret_cast<InstanceVoidMethodCallbackData*>(callbackInfo.Data());
2964    ✓✗         8    callbackInfo.SetData(callbackData->data);
2965 ✓✗✓✗          8    T* instance = Unwrap(callbackInfo.This().As<Object>());
       ✓✗
2966               8    auto cb = callbackData->callback;
2967 ✓✗✓✗          8    (instance->*cb)(callbackInfo);
2968              16    return nullptr;
2969              16  });
2970                }
2971
2972                template <typename T>
2973              12inline napi_value ObjectWrap<T>::InstanceMethodCallbackWrapper(
2974                    napi_env env,
2975                    napi_callback_info info) {
2976              12  return details::WrapCallback([&] {
2977 ✓✗✓✗         12    CallbackInfo callbackInfo(env, info);
2978                      InstanceMethodCallbackData* callbackData =
2979 ✓✗✓✗         12      reinterpret_cast<InstanceMethodCallbackData*>(callbackInfo.Data());
2980 ✓✗✓✗         12    callbackInfo.SetData(callbackData->data);
       ✓✗✓✗
2981 ✓✗✓✗         12    T* instance = Unwrap(callbackInfo.This().As<Object>());
       ✓✗✓✗
2982              12    auto cb = callbackData->callback;
       ✓✗✓✗
2983              24    return (instance->*cb)(callbackInfo);
       ✓✗✓✗
2984              24  });
2985                }
2986
2987                template <typename T>
2988              16inline napi_value ObjectWrap<T>::InstanceGetterCallbackWrapper(
2989                    napi_env env,
2990                    napi_callback_info info) {
2991              16  return details::WrapCallback([&] {
2992    ✓✗        16    CallbackInfo callbackInfo(env, info);
2993                      InstanceAccessorCallbackData* callbackData =
2994    ✓✗        16      reinterpret_cast<InstanceAccessorCallbackData*>(callbackInfo.Data());
2995    ✓✗        16    callbackInfo.SetData(callbackData->data);
2996 ✓✗✓✗         16    T* instance = Unwrap(callbackInfo.This().As<Object>());
       ✓✗
2997              16    auto cb = callbackData->getterCallback;
2998 ✓✗✓✗         32    return (instance->*cb)(callbackInfo);
2999              32  });
3000                }
3001
3002                template <typename T>
3003              12inline napi_value ObjectWrap<T>::InstanceSetterCallbackWrapper(
3004                    napi_env env,
3005                    napi_callback_info info) {
3006              12  return details::WrapCallback([&] {
3007    ✓✗        12    CallbackInfo callbackInfo(env, info);
3008                      InstanceAccessorCallbackData* callbackData =
3009    ✓✗        12      reinterpret_cast<InstanceAccessorCallbackData*>(callbackInfo.Data());
3010    ✓✗        12    callbackInfo.SetData(callbackData->data);
3011 ✓✗✓✗         12    T* instance = Unwrap(callbackInfo.This().As<Object>());
```

```
    ✓✗
3012              12    auto cb = callbackData->setterCallback;
3013 ✓✗✓✗         12    (instance->*cb)(callbackInfo, callbackInfo[0]);
    ✓✗
3014              24    return nullptr;
3015              24  });
3016                }
3017
3018                template <typename T>
3019               8inline void ObjectWrap<T>::FinalizeCallback(napi_env /*env*/, void* data, void* /*hint*/) {
3020               8  T* instance = reinterpret_cast<T*>(data);
3021 ✓✗✓✗          8  delete instance;
3022               8}
3023
3024                //////////////////////////////////////////////////////////////////////////////
3025                // HandleScope class
3026                //////////////////////////////////////////////////////////////////////////////
3027
3028                inline HandleScope::HandleScope(napi_env env, napi_handle_scope scope)
3029                    : _env(env), _scope(scope) {
3030                }
3031
3032             138inline HandleScope::HandleScope(Napi::Env env) : _env(env) {
3033             138  napi_status status = napi_open_handle_scope(_env, &_scope);
3034 ✗✓✗✗        138  NAPI_THROW_IF_FAILED(_env, status);
3035             138}
3036
3037             138inline HandleScope::~HandleScope() {
3038             138  napi_close_handle_scope(_env, _scope);
3039             138}
3040
3041                inline HandleScope::operator napi_handle_scope() const {
3042                  return _scope;
3043                }
3044
3045                inline Napi::Env HandleScope::Env() const {
3046                  return Napi::Env(_env);
3047                }
3048
3049                //////////////////////////////////////////////////////////////////////////////
3050                // EscapableHandleScope class
3051                //////////////////////////////////////////////////////////////////////////////
3052
3053                inline EscapableHandleScope::EscapableHandleScope(
3054                  napi_env env, napi_escapable_handle_scope scope) : _env(env), _scope(scope) {
3055                }
3056
3057         1000030inline EscapableHandleScope::EscapableHandleScope(Napi::Env env) : _env(env) {
3058         1000030  napi_status status = napi_open_escapable_handle_scope(_env, &_scope);
3059 ✗✓✗✗    1000030  NAPI_THROW_IF_FAILED(_env, status);
3060         1000030}
3061
3062         1000030inline EscapableHandleScope::~EscapableHandleScope() {
3063         1000030  napi_close_escapable_handle_scope(_env, _scope);
3064         1000030}
3065
3066                inline EscapableHandleScope::operator napi_escapable_handle_scope() const {
3067                  return _scope;
3068                }
3069
3070               2inline Napi::Env EscapableHandleScope::Env() const {
3071               2  return Napi::Env(_env);
3072                }
3073
3074              32inline Value EscapableHandleScope::Escape(napi_value escapee) {
3075                  napi_value result;
3076    ✓✗        32  napi_status status = napi_escape_handle(_env, _scope, escapee, &result);
3077 ✓✓✓✗        32  NAPI_THROW_IF_FAILED(_env, status, Value());
3078    ✓✗        31  return Value(_env, result);
3079                }
3080
3081                //////////////////////////////////////////////////////////////////////////////
3082                // AsyncWorker class
3083                //////////////////////////////////////////////////////////////////////////////
3084
3085                inline AsyncWorker::AsyncWorker(const Function& callback)
3086                  : AsyncWorker(callback, "generic") {
3087                }
3088
```

```
3089                inline AsyncWorker::AsyncWorker(const Function& callback,
3090                                                const char* resource_name)
3091                  : AsyncWorker(callback, resource_name, Object::New(callback.Env())) {
3092                }
3093
3094               2inline AsyncWorker::AsyncWorker(const Function& callback,
3095                                                const char* resource_name,
3096                                                const Object& resource)
3097    ✓✗         4  : AsyncWorker(Object::New(callback.Env()),
3098                                callback,
3099                                resource_name,
3100 ✓✗✓✗          2                resource) {
3101               2}
3102
3103                inline AsyncWorker::AsyncWorker(const Object& receiver,
3104                                                const Function& callback)
3105                  : AsyncWorker(receiver, callback, "generic") {
3106                }
3107
3108                inline AsyncWorker::AsyncWorker(const Object& receiver,
3109                                                const Function& callback,
3110                                                const char* resource_name)
3111                  : AsyncWorker(receiver,
3112                                callback,
3113                                resource_name,
3114                                Object::New(callback.Env())) {
3115                }
3116
3117               2inline AsyncWorker::AsyncWorker(const Object& receiver,
3118                                                const Function& callback,
3119                                                const char* resource_name,
3120                                                const Object& resource)
3121               2  : _env(callback.Env()),
3122                      _receiver(Napi::Persistent(receiver)),
3123 ✓✗✓✗ ✓✗       4    _callback(Napi::Persistent(callback)) {
3124                   napi_value resource_id;
3125                   napi_status status = napi_create_string_latin1(
3126    ✓✗         2      _env, resource_name, NAPI_AUTO_LENGTH, &resource_id);
3127 ✗✓✗✗          2  NAPI_THROW_IF_FAILED(_env, status);
3128
3129                   status = napi_create_async_work(_env, resource, resource_id, OnExecute,
3130 ✓✗✓✗          2                                  OnWorkComplete, this, &_work);
3131 ✗✓✗✗          2  NAPI_THROW_IF_FAILED(_env, status);
3132               2}
3133
3134               4inline AsyncWorker::~AsyncWorker() {
3135    ✓✗         2  if (_work != nullptr) {
3136               2    napi_delete_async_work(_env, _work);
3137               2    _work = nullptr;
3138                   }
3139    ✗✓         2}
3140
3141                inline AsyncWorker::AsyncWorker(AsyncWorker&& other) {
3142                  _env = other._env;
3143                  other._env = nullptr;
3144                  _work = other._work;
3145                  other._work = nullptr;
3146                  _receiver = std::move(other._receiver);
3147                  _callback = std::move(other._callback);
3148                  _error = std::move(other._error);
3149                }
3150
3151                inline AsyncWorker& AsyncWorker::operator =(AsyncWorker&& other) {
3152                  _env = other._env;
3153                  other._env = nullptr;
3154                  _work = other._work;
3155                  other._work = nullptr;
3156                  _receiver = std::move(other._receiver);
3157                  _callback = std::move(other._callback);
3158                  _error = std::move(other._error);
3159                  return *this;
3160                }
3161
3162                inline AsyncWorker::operator napi_async_work() const {
3163                  return _work;
3164                }
3165
3166                inline Napi::Env AsyncWorker::Env() const {
```

```
3167                 return Napi::Env(_env);
3168                }
3169
3170               2inline void AsyncWorker::Queue() {
3171               2  napi_status status = napi_queue_async_work(_env, _work);
3172 ✗✓✗✗          2  NAPI_THROW_IF_FAILED(_env, status);
3173               2}
3174
3175                inline void AsyncWorker::Cancel() {
3176                  napi_status status = napi_cancel_async_work(_env, _work);
3177                  NAPI_THROW_IF_FAILED(_env, status);
3178                }
3179
3180               2inline ObjectReference& AsyncWorker::Receiver() {
3181               2  return _receiver;
3182                }
3183
3184                inline FunctionReference& AsyncWorker::Callback() {
3185                  return _callback;
3186                }
3187
3188               1inline void AsyncWorker::OnOK() {
3189 ✓✗✓✗          1  _callback.MakeCallback(_receiver.Value(), std::initializer_list<napi_value>{});
    ✓✗
3190               1}
3191
3192               1inline void AsyncWorker::OnError(const Error& e) {
3193 ✓✗✓✗          1  _callback.MakeCallback(_receiver.Value(), std::initializer_list<napi_value>{ e.Value() });
    ✓✗✓✗
3194               1}
3195
3196               1inline void AsyncWorker::SetError(const std::string& error) {
3197               1  _error = error;
3198               1}
3199
3200               2inline void AsyncWorker::OnExecute(napi_env env, void* this_pointer) {
3201               2  AsyncWorker* self = static_cast<AsyncWorker*>(this_pointer);
3202                #ifdef NAPI_CPP_EXCEPTIONS
3203                  try {
3204    ✓✗          2    self->Execute();
3205                  } catch (const std::exception& e) {
3206                    self->SetError(e.what());
3207                  }
3208                #else // NAPI_CPP_EXCEPTIONS
3209                  self->Execute();
3210                #endif // NAPI_CPP_EXCEPTIONS
3211    ✗✗          2}
3212
3213               2inline void AsyncWorker::OnWorkComplete(
3214                    napi_env env, napi_status status, void* this_pointer) {
3215               2  AsyncWorker* self = static_cast<AsyncWorker*>(this_pointer);
3216    ✓✗          2  if (status != napi_cancelled) {
3217 ✓✗✓✗          2    HandleScope scope(self->_env);
3218               2    details::WrapCallback([&] {
3219    ✓✓          2      if (self->_error.size() == 0) {
3220               1        self->OnOK();
3221                      }
3222                      else {
3223    ✓✗          1        self->OnError(Error::New(self->_env, self->_error));
3224                      }
3225               2      return nullptr;
3226    ✓✗          2    });
3227                  }
3228    ✓✗          2  delete self;
3229               2}
3230
3231                ////////////////////////////////////////////////////////////////////////////////
3232                // Memory Management class
3233                ////////////////////////////////////////////////////////////////////////////////
3234
3235               3inline int64_t MemoryManagement::AdjustExternalMemory(Env env, int64_t change_in_bytes) {
3236                  int64_t result;
3237 ✓✗✓✗          3  napi_status status = napi_adjust_external_memory(env, change_in_bytes, &result);
3238 ✗✓✗✗          3  NAPI_THROW_IF_FAILED(env, status, 0);
    ✗✗
3239               3  return result;
3240                }
```

```
3241
3242                // These macros shouldn't be useful in user code.
3243                #undef NAPI_THROW
3244                #undef NAPI_THROW_IF_FAILED
3245
3246                } // namespace Napi
3247
3248                #endif // SRC_NAPI_INL_H_
```

Generated by: GCOVR (Version 3.4)