TypeScript

Language Specification

Version 0.9

June 2013

Table of Contents

1 Introduction

Web applications such as e-mail, maps, document editing, and collaboration tools are becoming an increasingly important part of the everyday computing. We designed TypeScript to meet the needs of the JavaScript programming teams that build and maintain large JavaScript programs such as web applications. TypeScript helps programming teams to define interfaces between software components and to gain insight into the behavior of existing JavaScript libraries. TypeScript also enables teams to reduce naming conflicts by organizing their code into dynamically-loadable modules. TypeScript's optional type system enables JavaScript programmers to use highly-productive development tools and practices: static checking, symbol-based navigation, statement completion, and code re-factoring.

TypeScript is a syntactic sugar for JavaScript. TypeScript syntax is a superset of Ecmascript 5 (ES5) syntax. Every JavaScript program is also a TypeScript program. The TypeScript compiler performs only file-local transformations on TypeScript programs and does not re-order variables declared in TypeScript. This leads to JavaScript output that closely matches the TypeScript input. TypeScript does not transform variable names, making tractable the direct debugging of emitted JavaScript. TypeScript optionally provides source maps, enabling source-level debugging. TypeScript tools typically emit JavaScript upon file save, preserving the test, edit, refresh cycle commonly used in JavaScript development.

TypeScript syntax includes several proposed features of Ecmascript 6 (ES6), including classes and modules. Classes enable programmers to express common object-oriented patterns in a standard way, making features like inheritance more readable and interoperable. Modules enable programmers to organize their code into components while avoiding naming conflicts. The TypeScript compiler provides module code generation options that support either static or dynamic loading of module contents.

TypeScript also provides to JavaScript programmers a system of optional type annotations. These type annotations are like the JSDoc comments found in the Closure system, but in TypeScript they are integrated directly into the language syntax. This integration makes the code more readable and reduces the maintenance cost of synchronizing type annotations with their corresponding variables.

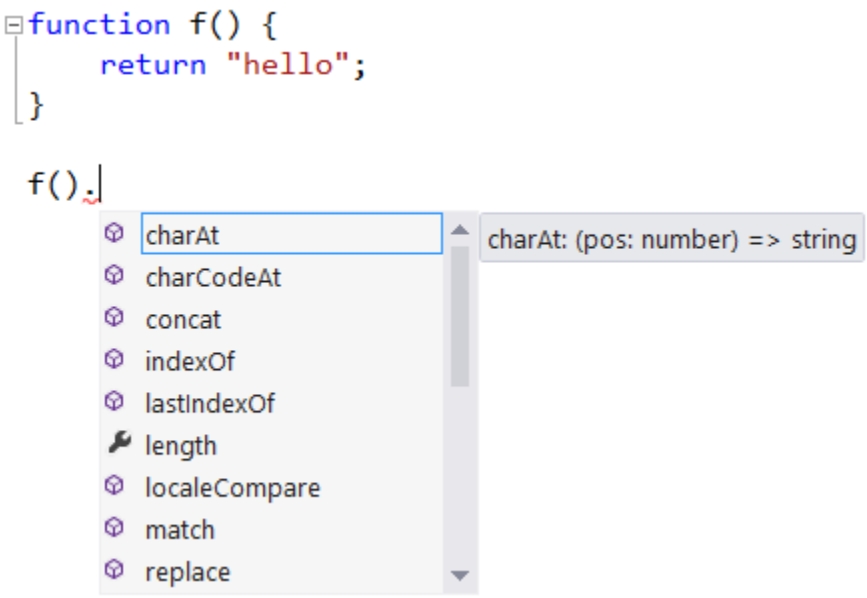
The TypeScript type system enables programmers to express limits on the capabilities of JavaScript objects, and to use tools that enforce these limits. To minimize the number of annotations needed for tools to become useful, the TypeScript type system makes extensive use of type inference. For example, from the following statement, TypeScript will infer that the variable 'i' has the type number.

```
var i = 0;
```

TypeScript will infer from the following function definition that the function f has return type string.

```
function f() {
    return "hello";
}
```

To benefit from this inference, a programmer can use the TypeScript language service. For example, a code editor can incorporate the TypeScript language service and use the service to find the members of a string object as in the following screen shot.



In this example, the programmer benefits from type inference without providing type annotations. Some beneficial tools, however, do require the programmer to provide type annotations. In TypeScript, we can express a parameter requirement as in the following code fragment.

```
function f(s: string) {
    return s;
}

f({});       // Error
f("hello");  // Ok
```

This optional type annotation on the parameter 's' lets the TypeScript type checker know that the programmer expects parameter 's' to be of type 'string'. Within the body of function 'f', tools can assume 's' is of type 'string' and provide operator type checking and member completion consistent with this assumption. Tools can also signal an error on the first call to 'f', because 'f' expects a string, not an object, as its parameter. For the function 'f', the TypeScript compiler will emit the following JavaScript code:

```
function f(s) {
    return s;
}
```

In the JavaScript output, all type annotations have been erased. In general, TypeScript erases all type information before emiting JavaScript.

1.1 Ambient Declarations

An ambient declaration introduces a variable into a TypeScript scope, but has zero impact on the emitted JavaScript program. Programmers can use ambient declarations to tell the TypeScript compiler that some other component will supply a variable. For example, by default the TypeScript compiler will print an error for uses of undefined variables. To add some of the common variables defined by browsers, a TypeScript programmer can use ambient declarations. The following example declares the 'document' object supplied by browsers. Because the declaration does not specify a type, the type 'any' is inferred. The type 'any' means that a tool can assume nothing about the shape or behavior of the document object. Some of the examples below will illustrate how programmers can use types to further characterize the expected behavior of an object.

```
declare var document;
document.title = "Hello";  // Ok because document has been declared
```

In the case of 'document', the TypeScript compiler automatically supplies a declaration, because TypeScript by default includes a file 'lib.d.ts' that provides interface declarations for the built-in JavaScript library as well as the Document Object Model.

The TypeScript compiler does not include by default an interface for jQuery, so to use jQuery, a programmer could supply a declaration such as:

```
declare var $;
```

Section 1.3 provides a more extensive example of how a programmer can add type information for jQuery and other libraries.

1.2 Function Types

Function expressions are a powerful feature of JavaScript. They enable function definitions to create closures: functions that capture information from the lexical scope surrounding the function's definition. Closures are currently JavaScript's only way of enforcing data encapsulation. By capturing and using environment variables, a closure can retain information that cannot be accessed from outside the closure. JavaScript programmers often use closures to express event handlers and other asynchronous callbacks, in which another software component, such as the DOM, will call back into JavaScript through a handler function.

TypeScript function types make it possible for programmers to express the expected *signature* of a function. A function signature is a sequence of parameter types plus a return type. The following example uses function types to express the callback signature requirements of an asynchronous voting mechanism.

```
function vote(candidate: string, callback: (result: string) => any) {
   // ...
}
```

```
vote("BigPig",
     function(result: string) {
         if (result === "BigPig") {
            // ...
         }
     }
);
```

In this example, the second parameter to 'vote' has the function type

```
(result: string) => any
```

which means the second parameter is a function returning type 'any' that has a single parameter of type 'string' named 'result'.

Section 3.7.2 provides additional information about function types.

1.3 Object Types

TypeScript programmers use *object types* to declare their expectations of object behavior. The following code uses an *object type literal* to specify the return type of the 'MakePoint' function.

```
var MakePoint: () => {
    x: number; y: number;
};
```

Programmers can give names to object types; we call named object types *interfaces*. For example, in the following code, an interface declares one required field (name) and one optional field (favoriteColor).

```
interface Friend {
    name: string;
    favoriteColor?: string;
}

function add(friend: Friend) {
    var name = friend.name;
}

add({ name: "Fred" });  // Ok
add({ favoriteColor: "blue" });  // Error, name required
add({ name: "Jill", favoriteColor: "green" });  // Ok
```

TypeScript object types model the diversity of behaviors that a JavaScript object can exhibit. For example, the jQuery library defines an object, '$', that has methods, such as 'get' (which sends an Ajax message), and fields, such as 'browser' (which gives browser vendor information). However, jQuery clients can also call '$' as a function. The behavior of this function depends on the type of parameters passed to the function.

The following code fragment captures a small subset of jQuery behavior, just enough to use jQuery in a simple way.

```
interface JQuery {
    text(content: string);
}

interface JQueryStatic {
    get(url: string, callback: (data: string) => any);
    (query: string): JQuery;
}

declare var $: JQueryStatic;

$.get("http://mysite.org/divContent",
      function (data: string) {
          $("div").text(data);
      }
);
```

The 'JQueryStatic' interface references another interface: 'JQuery'. This interface represents a collection of one or more DOM elements. The jQuery library can perform many operations on such a collection, but in this example the jQuery client only needs to know that it can set the text content of each jQuery element in a collection by passing a string to the 'text' method. The 'JQueryStatic' interface also contains a method, 'get', that performs an Ajax get operation on the provided URL and arranges to invoke the provided callback upon receipt of a response.

Finally, the 'JQueryStatic' interface contains a bare function signature

```
(query: string): JQuery;
```

The bare signature indicates that instances of the interface are callable. This example illustrates that TypeScript function types are just special cases of TypeScript object types. Specifically, function types are object types that contain only a call signature, but no properties. For this reason we can write any function type as an object type literal. The following example uses both forms to describe the same type.

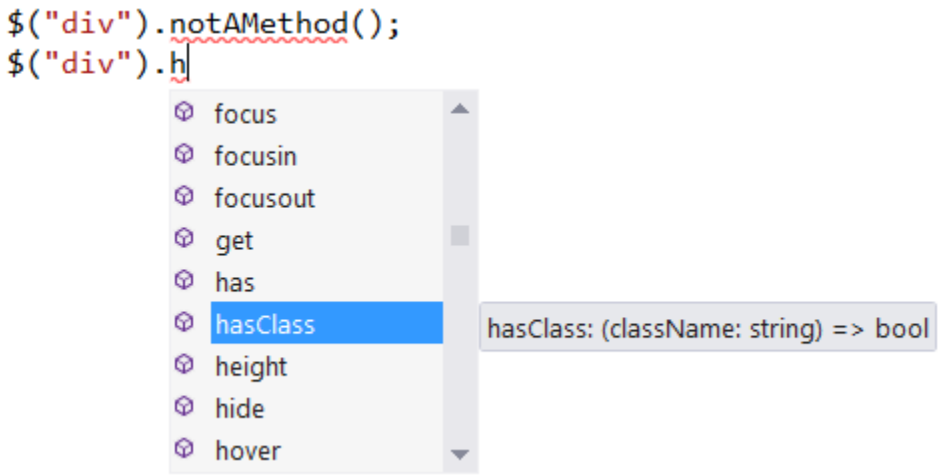
```
var f: { (): string; };
var sameType: () => string = f;     // Ok
var nope: () => number = sameType;  // Error: type mismatch
```

We mentioned above that the '$' function behaves differently depending on the type of its parameter. So far, our jQuery typing only captures one of these behaviors: return an object of type 'JQuery' when passed a string. To specify multiple behaviors, TypeScript supports *overloading* of function signatures in object types. For example, we can add an additional call signature to the 'JQueryStatic' interface.

```
(ready: () => any): any;
```

This signature denotes that a function may be passed as the parameter of the '$' function. When a function is passed to '$', the jQuery library will invoke that function when a DOM document is ready. Because TypeScript supports overloading, tools can use TypeScript to show all available function signatures with their documentation tips and to give the correct documentation once a function has been called with a particular signature.

A typical client would not need to add any additional typing but could just use a community-supplied typing to discover (through statement completion with documentation tips) and verify (through static checking) correct use of the library, as in the following screen shot.



Section 3.3 provides additional information about object types.

1.4 Structural Subtyping

Object types are compared *structurally*. For example, in the code fragment below, class 'CPoint' matches interface 'Point' because 'CPoint' has all of the required members of 'Point'. A class may optionally declare that it implements an interface, so that the compiler will check the declaration for structural compatibility. The example also illustrates that an object type can match the type inferred from an object literal, as long as the object literal supplies all of the required members.

```
interface Point {
    x: number;
    y: number;
}

function getX(p: Point) {
    return p.x;
}

class CPoint {
    constructor (public x: number, public y: number) { }
}
```

```
getX(new CPoint(0, 0));  // Ok, fields match

getX({ x: 0, y: 0, color: "red" });  // Extra fields Ok

getX({ x: 0 });  // Error: supplied parameter does not match
```

See Section 3.8 for more information about type comparisons.

1.5 Contextual Typing

Ordinarily, TypeScript type inference proceeds "bottom-up": from the leaves of an expression tree to its root. In the following example, TypeScript infers 'number' as the return type of the function 'mul' by flowing type information bottom up in the return expression.

```
function mul(a: number, b: number) {
    return a * b;
}
```

For variables and parameters without a type annotation or a default value, TypeScript infers type 'any', ensuring that compilers do not need non-local information about a function's call sites to infer the function's return type. Generally, this bottom-up approach provides programmers with a clear intuition about the flow of type information.

However, in some limited contexts, inference proceeds "top-down" from the context of an expression. Where this happens, it is called contextual typing. Contextual typing helps tools provide excellent information when a programmer is using a type but may not know all of the details of the type. For example, in the jQuery example, above, the programmer supplies a function expression as the second parameter to the 'get' method. During typing of that expression, tools can assume that the type of the function expression is as given in the 'get' signature and can provide a template that includes parameter names and types.

```
$.get("http://mysite.org/divContent",
      function (data) {
          $("div").text(data);  // TypeScript infers data is a string
      }
);
```

Contextual typing is also useful for writing out object literals. As the programmer types the object literal, the contextual type provides information that enables tools to provide completion for object member names.

Section 4.18 provides additional information about contextually typed expressions.

1.6 Classes

JavaScript practice has at least two common design patterns: the module pattern and the class pattern. Roughly speaking, the module pattern uses closures to hide names and to encapsulate private data, while

the class pattern uses prototype chains to implement many variations on object-oriented inheritance mechanisms. Libraries such as 'prototype.js' are typical of this practice.

This section and the module section below will show how TypeScript emits consistent, idiomatic JavaScript code to implement classes and modules that are closely aligned with the current ES6 proposal. The goal of TypeScript's translation is to emit exactly what a programmer would type when implementing a class or module unaided by a tool. This section will also describe how TypeScript infers a type for each class declaration. We'll start with a simple BankAccount class.

```
class BankAccount {
    balance = 0;
    deposit(credit: number) {
        this.balance += credit;
        return this.balance;
    }
}
```

This class generates the following JavaScript code.

```
var BankAccount = (function () {
    function BankAccount() {
        this.balance = 0;
    }
    BankAccount.prototype.deposit = function(credit) {
        this.balance += credit;
        return this.balance;
    };
    return BankAccount;
})();
```

This TypeScript class declaration creates a variable named 'BankAccount' whose value is the constructor function for 'BankAccount' instances. This declaration also creates an instance type of the same name. If we were to write this type as an interface it would look like the following.

```
interface BankAccount {
    balance: number;
    deposit(credit: number): number;
}
```

If we were to write out the function type declaration for the 'BankAccount' constructor variable, it would have the following form.

```
var BankAccount: new() => BankAccount;
```

The function signature is prefixed with the keyword 'new' indicating that the 'BankAccount' function must be called as a constructor. It is possible for a function's type to have both call and constructor signatures. For example, the type of the built-in JavaScript Date object includes both kinds of signatures.

If we want to start our bank account with an initial balance, we can add to the 'BankAccount' class a constructor declaration.

```
class BankAccount {
    balance: number;
    constructor(initially: number) {
        this.balance = initially;
    }
    deposit(credit: number) {
        this.balance += credit;
        return this.balance;
    }
}
```

This version of the 'BankAccount' class requires us to introduce a constructor parameter and then assign it to the 'balance' field. To simplify this common case, TypeScript accepts the following shorthand syntax.

```
class BankAccount {
    constructor(public balance: number) {
    }
    deposit(credit: number) {
        this.balance += credit;
        return this.balance;
    }
}
```

The 'public' keyword denotes that the constructor parameter is to be retained as a field. Public is the default visibility for class members, but a programmer can also specify private visibility for a class member. Private visibility is a design-time construct; it is enforced during static type checking but does not imply any runtime enforcement.

TypeScript classes also support inheritance, as in the following example.

```
class CheckingAccount extends BankAccount {
    constructor(balance: number) {
        super(balance);
    }
    writeCheck(debit: number) {
        this.balance -= debit;
    }
}
```

In this example, the class 'CheckingAccount' *derives* from class 'BankAccount'. The constructor for 'CheckingAccount' calls the constructor for class 'BankAccount' using the 'super' keyword. In the emitted JavaScript code, the prototype of 'CheckingAccount' will chain to the prototype of 'BankingAccount'.

TypeScript classes may also specify static members. Static class members become properties of the class constructor.

Section 8 provides additional information about classes.

1.7 Modules

Classes and interfaces support large-scale JavaScript development by providing a mechanism for describing how to use a software component that can be separated from that component's implementation. TypeScript enforces *encapsulation* of implementation in classes at design time (by restricting use of private members), but cannot enforce encapsulation at runtime because all object properties are accessible at runtime. Future versions of JavaScript may provide *private names* which would enable runtime enforcement of private members.

In the current version of JavaScript, the only way to enforce encapsulation at runtime is to use the module pattern: encapsulate private fields and methods using closure variables. The module pattern is a natural way to provide organizational structure and dynamic loading options by drawing a boundary around a software component. A module can also provide the ability to introduce namespaces, avoiding use of the global namespace for most software components.

The following example illustrates the JavaScript module pattern.

```
(function(exports) {
    var key = generateSecretKey();
    function sendMessage(message) {
        sendSecureMessage(message, key);
    }
    exports.sendMessage = sendMessage;
})(MessageModule);
```

This example illustrates the two essential elements of the module pattern: a *module closure* and a *module object*. The module closure is a function that encapsulates the module's implementation, in this case the variable 'key' and the function 'sendMessage'. The module object contains the exported variables and functions of the module. Simple modules may create and return the module object. The module above takes the module object as a parameter, 'exports', and adds the 'sendMessage' property to the module object. This *augmentation* approach simplifies dynamic loading of modules and also supports separation of module code into multiple files.

The example assumes that an outer lexical scope defines the functions 'generateSecretKey' and 'sendSecureMessage'; it also assumes that the outer scope has assigned the module object to the variable 'MessageModule'.

TypeScript modules provide a mechanism for succinctly expressing the module pattern. In TypeScript, programmers can combine the module pattern with the class pattern by nesting modules and classes within an outer module.

The following example shows the definition and use of a simple module.

```
module M {
    var s = "hello";
    export function f() {
        return s;
    }
}

M.f();
M.s;  // Error, s is not exported
```

In this example, variable 's' is a private feature of the module, but function 'f' is exported from the module and accessible to code outside of the module. If we were to describe the effect of module 'M' in terms of interfaces and variables, we would write

```
interface M {
    f(): string;
}

var M: M;
```

The interface 'M' summarizes the externally visible behavior of module 'M'. In this example, we can use the same name for the interface as for the initialized variable because in TypeScript type names and variable names do not conflict: each lexical scope contains a variable declaration space and type declaration space (see Section 2.3 for more details).

Module 'M' is an example of an *internal* module, because it is nested within the *global* module (see Section 9 for more details). The TypeScript compiler emits the following JavaScript code for this module.

```
var M;
(function(M) {
    var s = "hello";
    function f() {
        return s;
    }
    M.f = f;
})(M||(M={}));
```

In this case, the compiler assumes that the module object resides in global variable 'M', which may or may not have been initialized to the desired module object.

TypeScript also supports *external* modules, which are files that contain top-level *export* and *import* directives. For this type of module the TypeScript compiler will emit code whose module closure and module object implementation vary according to the specified dynamic loading system, for example, the Asynchronous Module Definition system.

2 Basic Concepts

The remainder of this document is the formal specification of the TypeScript programming language and is intended to be read as an adjunct to the ECMAScript Language Specification (specifically, the ECMA-262 Standard, 5th Edition). This document describes the syntactic grammar added by TypeScript along with the compile-time processing and type checking performed by the TypeScript compiler, but it only minimally discusses the run-time behavior of programs since that is covered by the ECMAScript specification.

2.1 Grammar Conventions

The syntactic grammar added by TypeScript language is specified throughout this document using the existing conventions and production names of the ECMAScript grammar. In places where TypeScript augments an existing grammar production it is so noted. For example:

CallExpression: *(Modified)*
…
`super` (*ArgumentList*$_{opt}$)
`super` `.` *IdentifierName*

The '*(Modified)*' annotation indicates that an existing grammar production is being replaced, and the '…' references the contents of the original grammar production.

2.2 Namespaces and Named Types

TypeScript supports ***named types*** that can be organized in hierarchical ***namespaces***. Namespaces are introduced by module declarations and named types are introduced by class, interface, and enum declarations. Named types are denoted by qualified names that extend from some root module (possibly the global module) to the point of their declaration. The example

```
module X {
    export module Y {
        export interface Z { }
    }
    export interface Y { }
}
```

declares two interface types with the qualified names 'X.Y.Z' and 'X.Y' relative to the root module in which 'X' is declared.

In a qualified type name all identifiers but the last one refer to namespaces and the last identifier refers to a named type. Named type and namespace names are in separate declaration spaces and it is therefore possible for a named type and a namespace to have the same name, as in the example above.

The hierarchy formed by namespace and named type names partially mirrors that formed by module instances and members. The example

```
module A {
    export module B {
        export class C { }
    }
}
```

introduces a named type with the qualified name 'A.B.C' and also introduces a constructor function that can be accessed using the expression 'A.B.C'. Thus, in the example

```
var c: A.B.C = new A.B.C();
```

the two occurrences of 'A.B.C' in fact refer to different entities. It is the context of the occurrences that determines whether 'A.B.C' is processed as a type name or an expression.

2.3 Declarations

Declarations introduce names in the ***declaration spaces*** to which they belong. It is an error to have two names with same spelling in the same declaration space. Declaration spaces exist as follows:

- The global module and each external or internal module has a declaration space for variables (including functions, modules, class constructor functions, and enum objects), a declaration space for named types (classes, interfaces, and enums), and a declaration space for namespaces (containers of named types). Every declaration (whether local or exported) in a module contributes to one or more of these declaration spaces.
- Each external or internal module has a declaration space for exported members, a declaration space for exported named types, and a declaration space for exported namespaces. All export declarations in the module contribute to these declaration spaces. Each internal module's export declaration spaces are shared with other internal modules that have the same root module and the same qualified name starting from that root module.
- Each class declaration has a declaration space for instance members, a declaration space for static members, and a declaration space for type parameters.
- Each interface declaration has a declaration space for members and a declaration space for type parameters. An interface's declaration space is shared with other interfaces that have the same root module and the same qualified name starting from that root module.
- Each enum declaration has a declaration space for its enum members. An enum's declaration space is shared with other enums that have the same root module and the same qualified name starting from that root module.
- Each function declaration (including constructor, member function, and member accessor declarations) and each function expression has a declaration space for variables (parameters, local variables, and local functions) and a declaration space for type parameters.
- Each object literal has a declaration space for its properties.
- Each object type literal has a declaration space for its members.

Top-level declarations in a source file with no top-level import or export declarations belong to the ***global module***. Top-level declarations in a source file with one or more top-level import or export declarations belong to the ***external module*** represented by that source file.

An internal module declaration contributes a namespace name (representing a container of types) and possibly a member name (representing the module instance) to the containing module. A class declaration contributes both a member name (representing the constructor function) and a type name (representing the class type) to the containing module. An interface declaration contributes a type name to the containing module. An enum declaration contributes both a member name (representing the enum object) and a type name (representing the enum type) to the containing module. Any other declaration contributes a member name to the declaration space to which it belongs.

The ***parent module*** of an entity is defined as follows:

- The parent module of an entity declared in an internal module is that internal module.
- The parent module of an entity declared in an external module is that external module.
- The parent module of an entity declared in the global module is the global module.
- The parent module of an external module is the global module.

The ***root module*** of an entity is defined as follows:

- The root module of a non-exported entity is the entity's parent module.
- The root module of an exported entity is the root module of the entity's parent module.

Intuitively, the root module of an entity is the outermost module body from within which the entity is reachable.

Interfaces, enums, and internal modules are "open ended," meaning that interface, enum, and internal module declarations with the same qualified name relative to a common root are automatically merged. For further details, see sections 7.2, 9.3, and 10.5.

Namespace, type, and member names exist in separate declaration spaces. Furthermore, declarations of non-instantiated modules (modules that contain only interfaces or modules at all levels of nesting) do not introduce a member name in their containing declaration space. This means that the following is permitted, provided module 'X' contains only interface or module declarations at all levels of nesting:

```
module M {
    module X { ... }      // Namespace
    interface X { ... }   // Type
    var X;                // Member
}
```

If module 'X' above was an instantiated module (section 10.1) it would cause a member 'X' to be introduced in 'M'. This member would conflict with the variable 'X' and thus cause an error.

Instance and static members in a class are likewise in separate declaration spaces. Thus the following is permitted:

```
class C {
    x: number;          // Instance member
    static x: string;   // Static member
}
```

2.4 Scopes

The ***scope*** of a name is the region of program text within which it is possible to refer to the entity declared by that name without qualification of the name. The scope of a name depends on the context in which the name is declared. The contexts are listed below in order from outermost to innermost:

- The scope of an entity declared in the global module is the entire program text.
- The scope of an entity declared in an external module is the source file of that external module.
- The scope of an exported entity declared in an internal module is the body of that module and every internal module with the same root and the same qualified name relative to that root.
- The scope of a non-exported entity declared within an internal module declaration is the body of that internal module declaration.
- The scope of a type parameter declared in a class or interface declaration is that entire declaration, including constraints, extends clause, implements clause, and declaration body, but not including static member declarations.
- The scope of a type parameter declared in a call or construct signature is that entire signature declaration, including constraints, parameter list, and return type. If the signature is part of a function implementation, the scope includes the function body.
- The scope of a parameter, local variable, or local function declared within a function declaration (including a constructor, member function, or member accessor declaration) or function expression is the body of that function declaration or function expression.

Scopes may overlap, for example through nesting of modules and functions. When the scopes of two entities with the same name overlap, the entity with the innermost declaration takes precedence and access to the outer entity is either not possible or only possible by qualifying its name.

When an identifier is resolved as a *TypeName* (section 3.6.2), only classes, interfaces, enums, and type parameters are considered and other entities in scope are ignored.

When an identifier is resolved as a *ModuleName* (section 3.6.2), only modules are considered and other entities in scope are ignored.

When an identifier is resolved as a *PrimaryExpression* (section 4.3), only instantiated modules (section 10.1), classes, enums, functions, variables, and parameters are considered and other entities in scope are ignored.

Note that class and enum members are never directly in scope—they can only be accessed by applying the dot ('.') operator to a class instance or enum object. This even includes members of the current instance in a constructor or member function, which are accessed by applying the dot operator to `this`.

As the rules above imply, locally declared entities in an internal module are closer in scope than exported entities declared in other module declarations for the same internal module. For example:

```
var x = 1;
module M {
    export var x = 2;
    console.log(x);     // 2
}
module M {
    console.log(x);     // 2
}
module M {
    var x = 3;
    console.log(x);     // 3
}
```

3 Types

TypeScript adds optional static types to JavaScript. Types are used to place static constraints on program entities such as functions, variables, and properties so that compilers and development tools can offer better verification and assistance during software development. TypeScript's *static* compile-time type system closely models the *dynamic* run-time type system of JavaScript, allowing programmers to accurately express the type relationships that are expected to exist when their programs run and have those assumptions pre-validated by the TypeScript compiler. TypeScript's type analysis occurs entirely at compile-time and adds no run-time overhead to program execution.

All types in TypeScript are subtypes of a single top type called the Any type. The `any` keyword references this type. The Any type is the one type that can represent *any* JavaScript value with no constraints. All other types are categorized as ***primitive types***, ***object types***, or ***type parameters***. These types introduce various static constraints on their values.

The primitive types are the Number, Boolean, String, Void, Null, and Undefined types along with user defined enum types. The `number`, `boolean`, `string`, and `void` keywords reference the Number, Boolean, String, and Void primitive types respectively. The Void type exists purely to indicate the absence of a value, such as in a function with no return value. It is not possible to explicitly reference the Null and Undefined types—only *values* of those types can be referenced, using the `null` and `undefined` literals.

The object types are all class, interface, array, and literal types. Class and interface types are introduced through class and interface declarations and are referenced by the name given to them in their declarations. Class and interface types may be ***generic types*** which have one or more type parameters. Literal types are written as object, array, function, or constructor type literals and are used to compose new types from other types.

Declarations of modules, classes, properties, functions, variables and other language entities associate types with those entities. The mechanism by which a type is formed and associated with a language entity depends on the particular kind of entity. For example, a module declaration associates the module with an anonymous type containing a set of properties corresponding to the exported variables and functions in the module, and a function declaration associates the function with an anonymous type containing a call signature corresponding to the parameters and return type of the function. Types can be associated with variables through explicit ***type annotations***, such as

```
var x: number;
```

or through implicit ***type inference***, as in

```
var x = 1;
```

which infers the type of 'x' to be the Number primitive type because that is the type of the value used to initialize 'x'.

3.1 The Any Type

The Any type is used to represent any JavaScript value. A value of the Any type supports the same operations as a value in JavaScript and minimal static type checking is performed for operations on Any values. Specifically, properties of any name can be accessed through an Any value and Any values can be called as functions or constructors with any argument list.

The `any` keyword references the Any type. In general, in places where a type is not explicitly provided and TypeScript cannot infer one, the Any type is assumed.

The Any type is a supertype of all types.

Some examples:

```
var x: any;             // Explicitly typed
var y;                  // Same as y: any
var z: { a; b; };       // Same as z: { a: any; b: any; }

function f(x) {         // Same as f(x: any): void
    console.log(x);
}
```

3.2 Primitive Types

The primitive types are the Number, Boolean, String, Void, Null, and Undefined types and all user defined enum types.

3.2.1 The Number Type

The Number primitive type corresponds to the similarly named JavaScript primitive type and represents double-precision 64-bit format IEEE 754 floating point values.

The `number` keyword references the Number primitive type and numeric literals may be used to write values of the Number primitive type.

For purposes of determining type relationships (section 3.8) and accessing properties (section 4.10), the Number primitive type behaves as an object type with the same properties as the global interface type 'Number'.

Some examples:

```
var x: number;          // Explicitly typed
var y = 0;              // Same as y: number = 0
var z = 123.456;        // Same as z: number = 123.456
var s = z.toFixed(2);   // Property of Number interface
```

3.2.2 The Boolean Type

The Boolean primitive type corresponds to the similarly named JavaScript primitive type and represents logical values that are either true or false.

The `boolean` keyword references the Boolean primitive type and the `true` and `false` literals reference the two Boolean truth values.

For purposes of determining type relationships (section 3.8) and accessing properties (section 4.10), the Boolean primitive type behaves as an object type with the same properties as the global interface type 'Boolean'.

Some examples:

```
var b: boolean;         // Explicitly typed
var yes = true;         // Same as yes: boolean = true
var no = false;         // Same as no: boolean = false
```

3.2.3 The String Type

The String primitive type corresponds to the similarly named JavaScript primitive type and represents sequences of characters stored as Unicode UTF-16 code units.

The `string` keyword references the String primitive type and string literals may be used to write values of the String primitive type.

For purposes of determining type relationships (section 3.8) and accessing properties (section 4.10), the String primitive type behaves as an object type with the same properties as the global interface type 'String'.

Some examples:

```
var s: string;          // Explicitly typed
var empty = "";         // Same as empty: string = ""
var abc = 'abc';        // Same as abc: string = "abc"
var c = abc.charAt(2);  // Property of String interface
```

3.2.4 The Void Type

The Void type, referenced by the `void` keyword, represents the absence of a value and is used as the return type of functions with no return value.

The only possible values for the Void type are `null` and `undefined`. The Void type is a subtype of the Any type and a supertype of the Null and Undefined types, but otherwise Void is unrelated to all other types.

NOTE: We might consider disallowing declaring variables of type Void as they serve no useful purpose. However, because Void is permitted as a type argument to a generic type or function it is not feasible to disallow Void properties or parameters.

3.2.5 The Null Type

The Null type corresponds to the similarly named JavaScript primitive type and is the type of the `null` literal.

The `null` literal references the one and only value of the Null type. It is not possible to directly reference the Null type itself.

The Null type is a subtype of all types, except the Undefined type. This means that `null` is considered a valid value for all primitive types, object types, and type parameters, including even the Number and Boolean primitive types.

Some examples:

```
var n: number = null;   // Primitives can be null
var x = null;           // Same as x: any = null
var e: Null;            // Error, can't reference Null type
```

3.2.6 The Undefined Type

The Undefined type corresponds to the similarly named JavaScript primitive type and is the type of the `undefined` literal.

The `undefined` literal denotes the value given to all uninitialized variables and is the one and only value of the Undefined type. It is not possible to directly reference the Undefined type itself.

The undefined type is a subtype of all types. This means that `undefined` is considered a valid value for all primitive types, object types, and type parameters.

Some examples:

```
var n: number;          // Same as n: number = undefined
var x = undefined;      // Same as x: any = undefined
var e: Undefined;       // Error, can't reference Undefined type
```

3.2.7 Enum Types

Enum types are distinct user defined subtypes of the Number primitive type. Enum types are declared using enum declarations (section 9.1) and referenced using type references (section 3.6.2).

Enum types are assignable to the Number primitive type, and vice versa, but different enum types are not assignable to each other.

3.2.8 String Literal Types

Specialized signatures (section 3.7.2.4) permit string literals to be used as types in parameter type annotations. String literal types are permitted only in that context and nowhere else.

All string literal types are subtypes of the String primitive type.

3.3 Object Types

The object types include references to class and interface types as well as anonymous object types created by a number of constructs such as object literals, function declarations, and module declarations. Object types are composed from properties, call signatures, construct signatures, and index signatures, collectively called members.

3.3.1 Named Type References

Type references (section 3.6.2) to class and interface types are classified as object types. Type references to generic class and interface types include type arguments that are substituted for the type parameters of the class or interface to produce an actual object type.

3.3.2 Array Types

Array types represent JavaScript arrays. Array types are type references (section 3.6.2) created from the generic interface type 'Array' in the global module. Array type literals (section 3.6.4) provide a shorthand notation for creating such references.

Array literals (section 4.6) may be used to create values of array types.

3.3.3 Anonymous Types

Several constructs in the TypeScript language introduce new anonymous object types:

- Function and constructor type literals (section 3.6.4).
- Object type literals (section 3.7).
- Object literals (section 4.5).
- Function expressions (section 4.9) and function declarations (6.1).
- Constructor function types created by class declarations (section 8.2.5).
- Module instance types created by module declarations (section 10.3).

3.3.4 Members

Every object type is composed from zero or more of the following kinds of members:

- ***Properties***, which define the names and types of the properties of objects of the given type. Property names are unique within their type.
- ***Call signatures***, which define the possible parameter lists and return types associated with applying call operations to objects of the given type.
- ***Construct signatures***, which define the possible parameter lists and return types associated with applying the `new` operator to objects of the given type.
- ***Index signatures***, which define type constraints for properties in the given type.

Properties are either ***public*** or ***private*** and are either ***required*** or ***optional***:

- Properties in a class declaration may be designated public or private, while properties declared in other contexts are always considered public. Private members are only accessible within the class

body containing their declaration, as described in section 8.2.2, and private properties match only themselves in subtype and assignment compatibility checks, as described in section 3.8.
- Properties in an object type literal or interface declaration may be designated required or optional, while properties declared in other contexts are always considered required. Properties that are optional in the target type of an assignment may be omitted from source objects, as described in section 3.8.3.

Call and construct signatures may be ***specialized*** (section 3.7.2.4) by including parameters with string literal types. Specialized signatures are used to express patterns where specific string values for some parameters cause the types of other parameters or the function result to become further specialized.

For purposes of determining type relationships (section 3.8) and accessing properties (section 4.10), object types appear to have certain additional members:

- Every object type appears to have the members of the global interface type 'Object' unless those members are hidden by members in the object type.
- An object type with one or more call or construct signatures appears to have the members of the global interface type 'Function' unless those members are hidden by members in the object type.

Object type members hide 'Object' or 'Function' interface members in the following manner:

- A property hides an 'Object' or 'Function' property with the same name.
- A call signature hides an 'Object' or 'Function' call signature with the same number of parameters and identical parameter types in the respective positions.
- A construct signature hides an 'Object' or 'Function' construct signature with the same number of parameters and identical parameter types in the respective positions.
- An index signature hides an 'Object' or 'Function' index signature with the same parameter type.

In effect, object types are subtypes of the 'Object' or 'Function' interface unless the object types define members that are incompatible with those of the 'Object' or 'Function' interface—which, for example, occurs if an object type defines a property with the same name as a property in the 'Object' or 'Function' interface but with a type that isn't a subtype of that in the 'Object' or 'Function' interface.

Some examples:

```
var o: Object = { x: 10, y: 20 };         // Ok
var f: Function = (x: number) => x * x;   // Ok
var err: Object = { toString: 0 };        // Error, incompatible toString
```

3.4 Type Parameters

A type parameter represents an actual type that the parameter is bound to in a generic type reference or a generic function call. Type parameters have constraints that establish upper bounds for their actual type arguments.

Since a type parameter represents a multitude of different type arguments, type parameters have certain restrictions compared to other types. In particular, a type parameter cannot be used as a base class or interface.

For purposes of determining type relationships (section 3.8), type parameters appear to be subtypes of the constraint specified in their declaration (or subtypes of 'Object' when no constraint was specified). Likewise, for purposes of accessing properties (section 4.10), type parameters appear to have the members of their declared constraint, but no other members.

3.5 Named Types

Class, interface, and enum types are ***named types*** that are introduced through class declarations (section 8.1), interface declarations (section 7.1), and enum declarations (9.1). Class and interface types may have type parameters and are then called ***generic types***. Conversely, named types without type parameters are called ***non-generic types***.

Interface declarations only introduce named types, whereas class declarations introduce named types *and* constructor functions that create instances of implementations of those named types. The named types introduced by class and interface declarations have only minor differences (classes can't declare optional members and interfaces can't declare private members) and are in most contexts interchangeable. In particular, class declarations with only public members introduce named types that function exactly like those created by interface declarations.

Named types are referenced through ***type references*** (section 3.6.2) that specify a type name and, if applicable, the type arguments to be substituted for the type parameters of the named type.

Named types are technically not types—only *references* to named types are. This distinction is particularly evident with generic types: Generic types are "templates" from which multiple *actual* types can be created by writing type references that supply type arguments to substitute in place of the generic type's type parameters. Only once this substitution takes place does a generic type denote an actual type.

TypeScript has a structural type system, and therefore a type created from a reference to a generic type is indistinguishable from an equivalent manually written expansion. For example, given the declaration

```
interface Pair<T1, T2> { first: T1; second: T2; }
```

the type reference

```
Pair<string, Entity>
```

is indistinguishable from the type

```
{ first: string; second: Entity; }
```

3.5.1 Type Parameter Lists

Class, interface, and function declarations may optionally include lists of type parameters enclosed in < and > brackets. Type parameters are also permitted in call signatures of object, function, and constructor type literals.

TypeParameters:
    < *TypeParameterList* >

TypeParameterList:
    TypeParameter
    TypeParameterList , *TypeParameter*

TypeParameter:
    Identifier *Constraint*$_{opt}$

Constraint:
    `extends` *TypeReference*

Type parameter names must be unique. A compile-time error occurs if two or more type parameters in the same *TypeParameterList* have the same name.

The scope of a type parameter extends over the entire declaration with which the type parameter list is associated, the only exception being static member declarations in classes.

Each type parameter has an associated type parameter ***constraint*** that establishes an upper bound for type arguments: A type argument for a given type parameter must be assignable to the type specified in the type parameter constraint. The type given in a type parameter constraint declaration must be an object type or a type parameter (i.e. a constraint cannot be the Any type or a primitive type). Omitting a constraint corresponds to specifying the global interface type 'Object'.

Type parameters may be referenced in type parameter constraints within the same type parameter list, including even constraint declarations that occur to the left of the type parameter.

3.5.2 Recursive Generic Types

Generic types are permitted to directly or indirectly reference themselves in a recursive fashion as long as such references do not generate an infinite series of new types. Specifically, within a generic type $G < T_1 , T_2 , \ldots , T_n >$ and the types referenced by it, it is an error to reference G with a type argument that wraps any of G's own type parameters (i.e. a type argument that wraps any T_x). A type parameter is said to be wrapped by a particular type when it is referenced, directly or indirectly, within that type.

Consider the following example:

```
interface List<T> {
    data: T;
    next: List<T>;
    owner: List<List<T>>;  // Error, recursive reference with wrapped T
}
```

In the example the 'owner' property creates an infinite series of new types that wrap a 'List<T>' around each previous 'List<T>'. Such generative recursion is prohibited by the rule above.

Note that it would be perfectly fine for the 'owner' property to have type 'List<List<Object>>' or any other type with a nested reference to 'List' that doesn't reference 'T'.

TODO: The restriction on generative recursion is likely to be removed.

3.5.3 Instance Types

Each named type has an associated actual type known as the ***instance type***. For a non-generic type, the instance type is simply a type reference to the non-generic type. For a generic type, the instance type is formed by creating a type reference from the generic type where each of the supplied type arguments is the corresponding type parameter. Since the instance type uses the type parameters it can be used only where the type parameters are in scope—that is, inside the declaration of the generic type. Within the constructor and member functions of a class, the type of `this` is the instance type of the class.

The following example illustrates the concept of an instance type:

```
class G<T> {               // Introduce type parameter T
    self: G<T>;            // Use T as type argument to form instance type
    f() {
        this.self = this;  // self and this are both of type G<T>
    }
}
```

3.6 Specifying Types

Types are specified either by referencing their keyword or name, by querying expression types, or by writing type literals which compose other types into new types.

Type:
- *PredefinedType*
- *TypeReference*
- *TypeQuery*
- *TypeLiteral*

3.6.1 Predefined Types

The `any`, `number`, `boolean`, `string`, and `void` keywords reference the Any type and the Number, Boolean, String, and Void primitive types respectively.

PredefinedType:
`any`
`number`
`boolean`
`string`
`void`

The predefined type keywords are reserved and cannot be used as names of user defined types.

3.6.2 Type References

A type reference references a named type or type parameter through its name and, in the case of a generic type, supplies a type argument list.

TypeReference:
TypeName *TypeArguments*$_{opt}$

TypeName:
Identifier
ModuleName `.` *Identifier*

ModuleName:
Identifier
ModuleName `.` *Identifier*

A *TypeReference* consists of a *TypeName* that a references a named type or type parameter. A reference to a generic type must be followed by a list of *TypeArguments*.

Resolution of a *TypeName* consisting of a single identifier is described in section 2.4.

Resolution of a *TypeName* of the form *M.N*, where *M* is a *ModuleName* and *N* is an *Identifier*, proceeds by first resolving the module name *M*. If the resolution of *M* is successful and the resulting module contains an exported named type *N*, then *M.N* refers to that member. Otherwise, *M.N* is undefined.

Resolution of a *ModuleName* consisting of a single identifier is described in section 2.4.

Resolution of a *ModuleName* of the form *M.N*, where *M* is a *ModuleName* and *N* is an *Identifier*, proceeds by first resolving the module name *M*. If the resolution of *M* is successful and the resulting module contains an exported module member *N*, then *M.N* refers to that member. Otherwise, *M.N* is undefined.

3.6.2.1 Type Arguments

A type reference to a generic type must include a list of type arguments enclosed in angle brackets and separated by commas.

TypeArguments:
`<` *TypeArgumentList* `>`

TypeArgumentList:
    TypeArgument
    TypeArgumentList `,` *TypeArgument*

TypeArgument:
    Type

A type reference to a generic type is required to specify exactly one type argument for each type parameter of the referenced generic type. Each type argument must be assignable to (section 3.8.3) the constraint of the corresponding type parameter or otherwise an error occurs. An example:

```
interface A { a: string; }

interface B extends A { b: string; }

interface C extends B { c: string; }

interface G<T, U extends B> {
    x: T;
    y: U;
}

var v1: G<A, C>;               // Ok
var v2: G<{ a: string }, C>;   // Ok, equivalent to G<A, C>
var v3: G<A, A>;               // Error, A not valid argument for U
var v4: G<G<A, B>, C>;         // Ok
var v5: G<any, any>;           // Ok
var v6: G<any>;                // Error, wrong number of arguments
var v7: G;                     // Error, no arguments
```

A type argument is simply a *Type* and may itself be a type reference to a generic type, as demonstrated by 'v4' in the example above.

A type reference to a generic type *G* designates a type wherein all occurrences of *G*'s type parameters have been replaced with the actual type arguments supplied in the type reference. For example, the declaration of 'v1' above is equivalent to:

```
var v1: {
    x: { a: string; }
    y: { a: string; b: string; c: string };
};
```

3.6.3 Type Queries

A type query obtains the type of an expression.

TypeQuery:
    `typeof` *TypeQueryExpression*

TypeQueryExpression:
Identifier
TypeQueryExpression `.` *IdentifierName*

A type query consists of the keyword `typeof` followed by an expression. The expression is restricted to a single identifier or a sequence of identifiers separated by periods. The expression is processed as an identifier expression (section 4.3) or property access expression (section 4.10), the type of which becomes the result. Similar to other static typing constructs, type queries are erased from the generated JavaScript code and add no run-time overhead.

Type queries are useful for capturing anonymous types that are generated by various constructs such as object literals, function declarations, and module declarations. For example:

```
var a = { x: 10, y: 20 };
var b: typeof a;
```

Above, 'b' is given the same type as 'a', namely '{ x: number; y: number; }'.

3.6.4 Type Literals

Type literals compose other types into new anonymous types.

TypeLiteral:
ObjectType
ArrayType
FunctionType
ConstructorType

ArrayType:
PredefinedType `[ ]`
TypeReference `[ ]`
ObjectType `[ ]`
ArrayType `[ ]`

FunctionType:
$TypeParameters_{opt}$ `(` $ParameterList_{opt}$ `) =>` *Type*

ConstructorType:
`new` $TypeParameters_{opt}$ `(` $ParameterList_{opt}$ `) =>` *Type*

Object type literals are the primary form of type literals and are described in section 3.7. Array, function, and constructor type literals are simply shorthand notations for other types:

Type literal	Equivalent form
T `[ ]`	`Array <` *T* `>`
`<` *TParams* `> (` *Params* `) =>` *Result*	`{ <` *TParams* `> (` *Params* `) :` *Result* `}`
`new <` *TParams* `> (` *Params* `) =>` *Result*	`{ new <` *TParams* `> (` *Params* `) :` *Result* `}`

As the table above illustrates, an array type literal is shorthand for a reference to the generic interface type 'Array' in the global module, a function type literal is shorthand for an object type containing a single call signature, and a constructor type literal is shorthand for an object type containing a single construct signature. Note that function and constructor types with multiple call or construct signatures cannot be written as function or constructor type literals but must instead be written as object type literals.

In order to avoid grammar ambiguities, array type literals permit only a restricted set of notations for the element type. Specifically, an A*rrayType* cannot start with a *TypeQuery*, *FunctionType*, or *ConstructorType*. To use one of those forms for the element type, an array type must be written using the 'Array<T>' notation. For example, the type

```
() => string[]
```

denotes a function returning a string array, not an array of functions returning string. The latter can be expressed using 'Array<T>' notation

```
Array<() => string>
```

or by writing the element type as an object type literal

```
{ (): string }[]
```

3.7 Object Type Literals

An object type literal defines an object type by specifying the set of members that are statically considered to be present in instances of the type. Object type literals can be given names using interface declarations but are otherwise anonymous.

ObjectType:
`{` *TypeBody*$_{opt}$ `}`

TypeBody:
TypeMemberList `;`$_{opt}$

TypeMemberList:
TypeMember
TypeMemberList `;` *TypeMember*

TypeMember:
PropertySignature
CallSignature
ConstructSignature
IndexSignature
MethodSignature

The members of an object type literal are specified as a combination of property, call, construct, index, and method signatures. The signatures are separated by semicolons and enclosed in curly braces.

3.7.1 Property Signatures

A property signature declares the name and type of a property member.

PropertySignature:
PropertyName ?$_{opt}$ *TypeAnnotation*$_{opt}$

PropertyName:
IdentifierName
StringLiteral
NumericLiteral

The *PropertyName* production, reproduced above from the ECMAScript grammar, permits a property name to be any identifier (including a reserved word), a string literal, or a numeric literal. String literals can be used to give properties names that are not valid identifiers, such as names containing blanks. Numeric literal property names are equivalent to string literal property names with the string representation of the numeric literal, as defined in the ECMAScript specification.

The *PropertyName* of a property signature must be unique within its containing type. If the property name is followed by a question mark, the property is optional. Otherwise, the property is required.

If a property signature omits a *TypeAnnotation*, the Any type is assumed.

3.7.2 Call Signatures

A call signature defines the type parameters, parameter list, and return type associated with applying a call operation (section 4.12) to an instance of the containing type. A type may ***overload*** call operations by defining multiple different call signatures.

CallSignature:
TypeParameters$_{opt}$ (*ParameterList*$_{opt}$) *TypeAnnotation*$_{opt}$

A call signature that includes *TypeParameters* (section 3.5.1) is called a ***generic call signature***. Conversely, a call signature with no *TypeParameters* is called a non-generic call signature.

As well as being members of object type literals, call signatures occur in method signatures (section 3.7.5), function expressions (section 4.9), and function declarations (section 6.1).

An object type containing call signatures is said to be a ***function type***.

It is an error for a type to declare multiple call signatures that are considered identical (section 3.8.1) or differ only in their return types.

3.7.2.1 Type Parameters

Type parameters in call signatures provide a mechanism for expressing the relationships of parameter and return types in call operations. For example, a signature might introduce a type parameter and use it as both a parameter type and a return type, in effect describing a function that returns a value of the same type as its argument.

The scope of a type parameter extends over the entire call signature in which the type parameter is introduced. Thus, type parameters may be referenced in type parameter constraints, parameter types, and return type annotations in their associated call signature.

Type arguments for call signature type parameters may be explicitly specified in a call operation or may, when possible, be inferred (section 4.12.2) from the types of the regular arguments in the call.

Some examples of call signatures with type parameters:

`<T>(x: T): T`	A function taking an argument of any type, returning a value of that same type.
`<T>(x: T, y: T): T[]`	A function taking two values of the same type, returning an array of that type.
`<T, U>(x: T, y: U): { x: T; y: U; }`	A function taking two arguments of different types, returning an object with properties 'x' and 'y' of those types.
`<T, U>(a: T[], f: (x: T) => U): U[]`	A function taking an array of one type and a function argument, returning an array of another type, where the function argument takes a value of the first array element type and returns a value of the second array element type.

3.7.2.2 Parameter List

A signature's parameter list consists of zero or more required parameters, followed by zero or more optional parameters, finally followed by an optional rest parameter.

ParameterList:
    RequiredParameterList
    OptionalParameterList
    RestParameter
    RequiredParameterList `,` *OptionalParameterList*
    RequiredParameterList `,` *RestParameter*
    OptionalParameterList `,` *RestParameter*
    RequiredParameterList `,` *OptionalParameterList* `,` *RestParameter*

RequiredParameterList:
    RequiredParameter
    RequiredParameterList `,` *RequiredParameter*

RequiredParameter:
    PublicOrPrivate$_{opt}$ *Identifier* *TypeAnnotation*$_{opt}$
    Identifier `:` *StringLiteral*

PublicOrPrivate:
    `public`
    `private`

OptionalParameterList:
    OptionalParameter
    OptionalParameterList `,` *OptionalParameter*

OptionalParameter:
    PublicOrPrivate$_{opt}$ *Identifier* `?` *TypeAnnotation*$_{opt}$
    PublicOrPrivate$_{opt}$ *Identifier* *TypeAnnotation*$_{opt}$ *Initialiser*

RestParameter:
    `...` *Identifier* *TypeAnnotation*$_{opt}$

Parameter names must be unique. A compile-time error occurs if two or more parameters have the same name.

A parameter is permitted to include a `public` or `private` modifier only if it occurs in the parameter list of a *ConstructorImplementation* (section 8.3.1).

A parameter with a type annotation is considered to be of that type. A type annotation for a rest parameter must denote an array type.

A parameter with no type annotation or initializer is considered to be of type `any`, unless it is a rest parameter, in which case it is considered to be of type `any[]`.

When a parameter type annotation specifies a string literal type, the containing signature is a specialized signature (section 3.7.2.4). Specialized signatures are not permitted in conjunction with a function body,

i.e. the *FunctionExpression*, *FunctionImplementation*, *MemberFunctionImplementation*, and *ConstructorImplementation* grammar productions do not permit parameters with string literal types.

A parameter can be marked optional by following its name with a question mark (`?`) or by including an initializer. The form that includes an initializer is permitted only in conjunction with a function body, i.e. only in a *FunctionExpression*, *FunctionImplementation*, *MemberFunctionImplementation*, or *ConstructorImplementation* grammar production.

TODO: Rest parameters.

3.7.2.3 Return Type

If present, a call signature's return type annotation specifies the type of the value computed and returned by a call operation. A `void` return type annotation is used to indicate that a function has no return value.

When a call signature with no return type annotation occurs in a context without a function body, the return type is assumed to be the Any type.

When a call signature with no return type annotation occurs in a context that has a function body (specifically, a function implementation, a member function implementation, or a member accessor declaration), the return type is inferred from the function body as described in section 6.3.

3.7.2.4 Specialized Signatures

When a parameter type annotation specifies a string literal type (section 3.2.8), the containing signature is considered a specialized signature. Specialized signatures are used to express patterns where specific string values for some parameters cause the types of other parameters or the function result to become further specialized. For example, the declaration

```
interface Document {
    createElement(tagName: string): HTMLElement;
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
}
```

states that calls to 'createElement' with the string literals "div", "span", and "canvas" return values of type 'HTMLDivElement', 'HTMLSpanElement', and 'HTMLCanvasElement' respectively, and that calls with all other string expressions return values of type 'HTMLElement'. Because string literal types are subtypes of the String primitive type, when a function call argument matches a parameter of a string literal type in a specialized signature, the overload resolution rules (section 4.12.1) give preference to that signature over a similar signature with a regular string parameter.

Every specialized call or construct signature in an object type must be a subtype of at least one non-specialized call or construct signature in the same object type. For example, the 'createElement' property in the example above is of a type that contains three specialized signatures, all of which are subtypes of the non-specialized signature in the type.

3.7.3 Construct Signatures

A construct signature defines the parameter list and return type associated with applying the `new` operator (section 4.11) to an instance of the containing type. A type may overload `new` operations by defining multiple construct signatures with different parameter lists.

ConstructSignature:
`new` *TypeParameters*$_{opt}$ `(` *ParameterList*$_{opt}$ `)` *TypeAnnotation*$_{opt}$

The type parameters, parameter list, and return type of a construct signature are subject to the same rules as a call signature.

A type containing construct signatures is said to be a ***constructor type***.

It is an error for a type to declare multiple construct signatures that are considered identical (section 3.8.1) or differ only by their return types.

3.7.4 Index Signatures

An index signature defines a type constraint for properties in the containing type.

IndexSignature:
`[` *Identifier* `:` `string` `]` *TypeAnnotation*
`[` *Identifier* `:` `number` `]` *TypeAnnotation*

There are two kinds of index signatures:

- ***String index signatures***, specified using index type `string`, define type constraints for all properties and numeric index signatures in the containing type. Specifically, in a type with a string index signature of type *T*, all properties and numeric index signatures must have types that are subtypes of *T*.
- ***Numeric index signatures***, specified using index type `number`, define type constraints for all numerically named properties in the containing type. Specifically, in a type with a numeric index signature of type *T*, all numerically named properties must have types that are subtypes of *T*.

A ***numerically named property*** is a property whose name is a valid numeric literal. Specifically, a property with a name *N* for which ToNumber(*N*) is not NaN, where ToNumber is the abstract operation defined in ECMAScript specification.

Index signatures affect the determination of the type that results from applying a bracket notation property access to an instance of the containing type, as described in section 4.10.

3.7.5 Method Signatures

A method signature is shorthand for declaring a property of a function type.

MethodSignature:
PropertyName `?`$_{opt}$ *CallSignature*

If the identifier is followed by a question mark, the property is optional. Otherwise, the property is required. Only object type literals and interfaces can declare optional properties.

A method signature of the form

PropName `<` *TypeParamList* `>` `(` *ParamList* `)` `:` *ReturnType*

is equivalent to the property declaration

PropName `:` `{` `<` *TypeParamList* `>` `(` *ParamList* `)` `:` *ReturnType* `}`

A literal type may ***overload*** a method by declaring multiple method signatures with the same name but differing parameter lists. Overloads must either all be required (question mark omitted) or all be optional (question mark included). A set of overloaded method signatures correspond to a declaration of a single property with a type composed from an equivalent set of call signatures. Specifically

PropName `<` $TypeParamList_1$ `>` `(` $ParamList_1$ `)` `:` $ReturnType_1$ `;`
PropName `<` $TypeParamList_2$ `>` `(` $ParamList_2$ `)` `:` $ReturnType_2$ `;`
…
PropName `<` $TypeParamList_n$ `>` `(` $ParamList_n$ `)` `:` $ReturnType_n$ `;`

is equivalent to

PropName `:` `{`
`<` $TypeParamList_1$ `>` `(` $ParamList_1$ `)` `:` $ReturnType_1$ `;`
`<` $TypeParamList_2$ `>` `(` $ParamList_2$ `)` `:` $ReturnType_2$ `;`
…
`<` $TypeParamList_n$ `>` `(` $ParamList_n$ `)` `:` $ReturnType_n$ `;` `}`

In the following example of an object type

```
{
    func1(x: number): number;          // Method signature
    func2: (x: number) => number;      // Function type literal
    func3: { (x: number): number };    // Object type literal
}
```

the properties ‘func1’, ‘func2’, and ‘func3’ are all of the same type, namely an object type with a single call signature taking a number and returning a number. Likewise, in the object type

```
{
    func4(x: number): number;
    func4(s: string): string;
    func5: {
        (x: number): number;
        (s: string): string;
    };
}
```

the properties ‘func4’ and ‘func5’ are of the same type, namely an object type with two call signatures taking and returning number and string respectively.

3.8 Type Relationships

Types in TypeScript have identity, subtype, supertype, and assignment compatibility relationships as defined in the following sections.

For purposes of determining type relationships, all object types appear to have the members of the ‘Object’ interface unless those members are hidden by members with the same name in the object types, and object types with one or more call or construct signatures appear to have the members of the ‘Function’ interface unless those members are hidden by members with the same name in the object types.

For purposes of determining subtype, supertype, and assignment compatibility relationships, the Number, Boolean, and String primitive types are treated as object types with the same properties as the ‘Number’, ‘Boolean’, and ‘String’ interfaces respectively. Likewise, enum types are treated as object types with the same properties as the ‘Number’ interface.

All type parameters appear to have the members of their constraint (or the ‘Object’ interface if they have no constraint), but no other members.

3.8.1 Type and Member Identity

Two types are considered ***identical*** when

- they are the same primitive type,
- they are the same type parameter, or
- they are object types with identical sets of members.

Two members are considered identical when

- they are public properties with identical names, optionality, and types,
- they are private properties originating in the same declaration and having identical types,
- they are identical call signatures,
- they are identical construct signatures, or
- they are index signatures of identical kind with identical types.

Two call or construct signatures are considered identical when they have the same number of type parameters and, considering those type parameters pairwise identical, have identical type parameter constraints, identical number of parameters of identical kinds and types, and identical return types.

Note that, except for primitive types and classes with private members, it is structure, not naming, of types that determines identity. Also, note that parameter names are not significant when determining identity of signatures.

Classes and interfaces can reference themselves in their internal structure, in effect creating recursive types with infinite nesting. For example, the type

```
interface A { next: A; }
```

contains an infinitely nested sequence of 'next' properties. Types such as this are perfectly valid but require special treatment when determining type relationships. Specifically, when comparing references to two named types *S* and *T* for a given relationship (identity, subtype, or assignability), the relationship in question is assumed to be true for every directly or indirectly nested occurrence of references to the same *S* and *T* (where same means originating in the same declaration). For example, consider the identity relationship between 'A' above and 'B' below:

```
interface B { next: C; }

interface C { next: D; }

interface D { next: B; }
```

To determine whether 'A' and 'B' are identical, first the 'next' properties of type 'A' and 'C' are compared. That leads to comparing the 'next' properties of type 'A' and 'D', which leads to comparing the 'next' properties of type 'A' and 'B'. Since 'A' and 'B' are already being compared this relationship is by definition true. That in turn causes the other comparisons to be true, and therefore the final result is true.

When this same technique is used to compare generic type references, two type references are considered the same when they originate in the same declaration and have identical type arguments. However, certain recursive generic patterns are prohibited, as explained in section 3.5.2.

Private properties match only if they originate in the same declaration and have identical types. Two distinct types might contain properties that originate in the same declaration if the types are separate parameterized references to the same generic class. In the example

```
class C<T> { private x: T; }

interface X { f(): string; }

interface Y { f(): string; }

var a: C<X>;
var b: C<Y>;
```

the variables 'a' and 'b' are of identical types because the two type references to 'C' create types with a private member 'x' that originates in the same declaration, and because the two private 'x' members have types with identical sets of members once the type arguments 'X' and 'Y' are substituted.

3.8.2 Subtypes and Supertypes

Given a type *S* and a substitution type *S'* where

- when *S* is the primitive type Number, Boolean, or String, *S′* is the global interface type 'Number', 'Boolean', or 'String',
- when *S* is an enum type, *S′* is the global interface type 'Number',
- when *S* is a type parameter, *S′* is the constraint of that type parameter,
- otherwise, *S′* is *S*,

S is a ***subtype*** of a type *T*, and *T* is a ***supertype*** of *S*, if one of the following is true:

- *S* and *T* are identical types.
- *T* is the Any type.
- *S* is the Undefined type.
- *S* is the Null type and *T* is not the Undefined type.
- *S* is an enum type and *T* is the primitive type Number.
- *S* is a string literal type and *T* is the primitive type String.
- *S′* and *T* are object types and, for each member *M* in *T*, one of the following is true:
    - *M* is a public property and *S′* contains a public property of the same name as *M* and a type that is a subtype of that of *M*.
    - *M* is a private property and *S′* contains a private property that originates in the same declaration as *M* and has a type that is a subtype of that of *M*.
    - *M* is an optional property and *S′* contains no property of the same name as *M*.
    - *M* is a non-specialized call or construct signature and *S′* contains a call or construct signature *N* where, when substituting 'Object' for all type parameters declared by *M* and *N* (if any),
        - the signatures are of the same kind (call or construct),
        - the number of non-optional parameters in *N* is less than or equal to that of *M*,
        - for parameter positions that are present in both signatures, each parameter type in *N* is a subtype or supertype of the corresponding parameter type in *M*,
        - the result type of *M* is Void, or the result type of *N* is a subtype of that of *M*.
    - *M* is a string index signature of type *U* and *S′* contains a string index signature of a type that is a subtype of *U*.
    - *M* is a numeric index signature of type *U* and *S′* contains a string or numeric index signature of a type that is a subtype of *U*.

When comparing call or construct signatures, parameter names are ignored and rest parameters correspond to an unbounded expansion of optional parameters of the rest parameter element type.

Note that specialized call and construct signatures (section 3.7.2.4) are not significant when determining subtype and supertype relationships.

3.8.3 Assignment Compatibility

Types are required to be assignment compatible in certain circumstances, such as expression and variable types in assignment statements and argument and parameter types in function calls.

Given a type *S* and a substitution type *S′* where

- when *S* is the primitive type Number, Boolean, or String, *S'* is the global interface type 'Number', 'Boolean', or 'String',
- when *S* is an enum type, *S'* is the global interface type 'Number',
- when *S* is a type parameter, *S'* is the constraint of that type parameter,
- otherwise, *S'* is *S*,

S is ***assignable to*** a type *T*, and *T* is ***assignable from*** *S*, if one of the following is true:

- *S* and *T* are identical types.
- *S* or *T* is the Any type.
- *S* is the Undefined type.
- *S* is the Null type and *T* is not the Undefined type.
- *S* or *T* is an enum type and the other is the primitive type Number.
- *S'* and *T* are object types and, for each member *M* in *T*, one of the following is true:
    - *M* is a public property and *S'* contains a public property of the same name as *M* and a type that is assignable to that of *M*.
    - *M* is a private property and *S'* contains a private property that originates in the same declaration as *M* and has a type that is assignable to that of *M*.
    - *M* is an optional property and *S'* contains no property of the same name as *M*.
    - *M* is a non-specialized call or construct signature and *S'* contains a call or construct signature *N* where, when substituting 'Object' for all type parameters declared by *M* and *N* (if any),
        - the signatures are of the same kind (call or construct),
        - the number of non-optional parameters in *N* is less than or equal to that of *M*,
        - for parameter positions that are present in both signatures, each parameter type in *N* is assignable to or from the corresponding parameter type in *M*,
        - the result type of *M* is Void, or the result type of *N* is assignable to that of *M*.
    - *M* is a string index signature of type *U* and *S'* contains a string index signature of a type that is assignable to *U*.
    - *M* is a numeric index signature of type *U* and *S'* contains a string or numeric index signature of a type that is assignable to *U*.

When comparing call or construct signatures, parameter names are ignored and rest parameters correspond to an unbounded expansion of optional parameters of the rest parameter element type.

Note that specialized call and construct signatures (section 3.7.2.4) are not significant when determining assignment compatibility.

The assignment compatibility and subtyping rules differ only in that

- the Any type is assignable to, but not a subtype of, all types, and
- the primitive type Number is assignable to, but not a subtype of, all enum types.

The assignment compatibility rules imply that, when assigning values or passing parameters, optional properties must either be present and of a compatible type, or not be present at all. For example:

```
function foo(x: { id: number; name?: string; }): void;

foo({ id: 1234 });                // Ok
foo({ id: 1234, name: "hello" }); // Ok
foo({ id: 1234, name: false });   // Error, name of wrong type
foo({ name: "hello" });           // Error, id required but missing
```

3.9 Widened Types

In several situations TypeScript infers types from context, alleviating the need for the programmer to explicitly specify types that appear obvious. For example

```
var name = "Steve";
```

infers the type of 'name' to be the String primitive type since that is the type of the value used to initialize it. When inferring the type of a variable, property or function result from an expression, the ***widened*** form of the source type is used as the inferred type of the target. The widened form of a type is the type in which all occurrences of the Null and Undefined types have been replaced with the type `any`.

The following example shows the results of widening types to produce inferred variable types.

```
var a = null;                   // var a: any
var b = undefined;              // var b: any
var c = { x: 0, y: null };      // var c: { x: number, y: any }
var d = [ null, undefined ];    // var d: any[]
```

3.10 Best Common Type

In some cases a ***best common type*** needs to be inferred from a set of types. In particular, return types of functions with multiple return statements and element types of array literals are found this way.

For an empty set of types, the best common type is the Any type.

For a non-empty set of types { T_1, T_2, ..., T_n }, the best common type is the one T_x in the set that is a supertype of every T_n. It is possible that no such type exists, in which case the best common type is an empty object type (the type `{}`).

4 Expressions

This chapter describes the manner in which TypeScript provides type inference and type checking for JavaScript expressions. TypeScript's type analysis occurs entirely at compile-time and adds no run-time overhead to expression evaluation.

TypeScript's typing rules define a type for every expression construct. For example, the type of the literal 123 is the Number primitive type, and the type of the object literal { a: 10, b: "hello" } is { a: number; b: string; }. The sections in this chapter describe these rules in detail.

In addition to type inference and type checking, TypeScript adds the following expression constructs:

- Optional parameter and return type annotations in function expressions.
- Default parameter values and rest parameters in function expressions.
- Arrow function expressions.
- Super calls and member access.
- Type assertions.

Unless otherwise noted in the sections that follow, TypeScript expressions and the JavaScript expressions generated from them are identical.

4.1 Values and References

Expressions are classified as ***values*** or ***references***. References are the subset of expressions that are permitted as the target of an assignment. Specifically, references are combinations of identifiers (section 4.3), parentheses (section 4.7), and property accesses (section 4.10). All other expression constructs described in this chapter are classified as values.

4.2 The this Keyword

The type of `this` in an expression depends on the location in which the reference takes place:

- In a constructor, member function, or member accessor, `this` is of the class instance type of the containing class.
- In a static function or static accessor, `this` is of a type corresponding to the constructor function type of the containing class without any construct signatures.
- In a function declaration or a standard function expression, `this` is of type Any.
- In the global module, `this` is of type Any.

In all other contexts it is a compile-time error to reference `this`.

In the body of an arrow function expression, references to `this` are rewritten in the generated JavaScript code, as described in section 4.9.2.

4.3 Identifiers

When an expression is an *Identifier*, the expression refers to the most nested module, class, enum, function, variable, or parameter with that name whose scope (section 2.4) includes the location of the reference.

The type of the expression is the type associated with the referenced entity:

- For a module, the object type associated with the module instance.
- For a class, the constructor type associated with the constructor function object.
- For an enum, the object type associated with the enum object.
- For a function, the function type associated with the function object.
- For a variable, the type of the variable.
- For a parameter, the type of the parameter.

In all cases the expression is classified as a reference.

4.4 Literals

Literals are typed as follows:

- The type of the `null` literal is the Null primitive type.
- The type of the literals `true` and `false` is the Boolean primitive type.
- The type of numeric literals is the Number primitive type.
- The type of string literals is the String primitive type.
- The type of regular expression literals is the global interface type 'RegExp'.

4.5 Object Literals

Object literals are extended to support type annotations in get and set accessors.

PropertyAssignment: (Modified)
PropertyName `:` *AssignmentExpression*
PropertyName *CallSignature* `{` *FunctionBody* `}`
GetAccessor
SetAccessor

GetAccessor:
`get` *PropertyName* `(` `)` $TypeAnnotation_{opt}$ `{` *FunctionBody* `}`

SetAccessor:
`set` *PropertyName* `(` *Identifier* $TypeAnnotation_{opt}$ `)` `{` *FunctionBody* `}`

The type of an object literal is an object type with the set of properties specified by the property assignments in the object literal. A get and set accessor may specify the same property name, but otherwise it is an error to specify multiple property assignments for the same property.

A property assignment of the form

PropertyName *CallSignature* `{` *FunctionBody* `}`

is simply shorthand for

PropertyName `:` `function` *CallSignature* `{` *FunctionBody* `}`

Each property assignment in an object literal is processed as follows:

- If the object literal is contextually typed and the contextual type contains a property with a matching name, the property assignment is contextually typed by the type of that property.
- Otherwise, if the object literal is contextually typed, the contextual type contains a numeric index signature, and the property assignment specifies a numeric property name, the property assignment is contextually typed by the type of the numeric index signature.
- Otherwise, if the object literal is contextually typed and the contextual type contains a string index signature, the property assignment is contextually typed by the type of the string index signature.
- Otherwise, the property assignment is processed without a contextual type.

The type of a property introduced by a property assignment of the form *Name* `:` *Expr* is the widened form (section 3.9) of the type of *Expr*.

A get accessor declaration is processed in the same manner as an ordinary function declaration (section 6.1) with no parameters. A set accessor declaration is processed in the same manner as an ordinary function declaration with a single parameter and a Void return type. When both a get and set accessor is declared for a property:

- If both accessors include type annotations, the specified types must be identical.
- If only one accessor includes a type annotation, the other behaves as if it had the same type annotation.
- If neither accessor includes a type annotation, the inferred return type of the get accessor becomes the parameter type of the set accessor.

If a get accessor is declared for a property, the return type of the get accessor becomes the type of the property. If only a set accessor is declared for a property, the parameter type (which may be type Any if no type annotation is present) of the set accessor becomes the type of the property.

When an object literal is contextually typed by a type that includes a string index signature of type *T*, the resulting type of the object literal includes a string index signature with the best common type of *T* and the types of the properties declared in the object literal. Likewise, when an object literal is contextually typed by a type that includes a numeric index signature of type *T*, the resulting type of the object literal includes a numeric index signature with the best common type of *T* and the types of the numerically named properties (section 3.7.4) declared in the object literal.

4.6 Array Literals

In the absence of a contextual type, the type of an array literal is *C*`[]`, where *C* is the best common type of the element expressions. Note that the type of an empty array literal is `any[]` since the Any type is the best common type of an empty set of types.

When an array literal is contextually typed (section 4.18) by an object type containing a numeric index signature of type *T*, each element expression is contextually typed by *T* and the type of the array literal is the best common type of *T* and the types of the element expressions.

4.7 Parentheses

A parenthesized expression

(*Expression*)

has the same type and classification as the *Expression* itself. Specifically, if the contained expression is classified as a reference, so is the parenthesized expression.

4.8 The super Keyword

The `super` keyword can be used in expressions to reference base class properties and the base class constructor.

CallExpression: (Modified)
...
`super` (*ArgumentList*$_{opt}$)
`super` `.` *IdentifierName*

4.8.1 Super Calls

Super calls consist of the keyword `super` followed by an argument list enclosed in parentheses. Super calls are only permitted in constructors of derived classes, as described in section 8.3.2.

A super call invokes the constructor of the base class on the instance referenced by `this`. A super call is processed as a function call (section 4.12) using the construct signatures of the base class constructor function type as the initial set of candidate signatures for overload resolution. Type arguments cannot be explicitly specified in a super call. If the base class is a generic class, the type arguments used to process a super call are always those specified in the `extends` clause that references the base class.

The type of a super call expression is Void.

The JavaScript code generated for a super call is specified in section 8.5.2.

4.8.2 Super Property Access

A super property access consists of the keyword `super` followed by a dot and an identifier. Super property accesses are used to access base class member functions from derived classes.

Super property accesses are permitted as follows:

- In a constructor, instance member function, or instance member accessor of a derived class, a super property access must specify a public instance member function of the base class.
- In a static member function or static member accessor of a derived class, a super property access must specify a public static member function of the base class.

Super property accesses are not permitted in other contexts, and it is not possible to access other kinds of base class members in a super property access.

Super property accesses are typically used to access overridden base class member functions from derived class member functions. For an example of this, see section 8.4.2.

The JavaScript code generated for a super property access is specified in section 8.5.2.

4.9 Function Expressions

Function expressions are extended from JavaScript to optionally include parameter and return type annotations, and a new compact form, called arrow function expressions, is introduced.

FunctionExpression: (Modified)
`function` *Identifier*$_{opt}$ *CallSignature* `{` *FunctionBody* `}`

AssignmentExpression: (Modified)
…
ArrowFunctionExpression

ArrowFunctionExpression:
ArrowFormalParameters `=>` *Block*
ArrowFormalParameters `=>` *AssignmentExpression*

ArrowFormalParameters:
CallSignature
Identifier

The terms ***standard function expression*** and ***arrow function expression*** are used to refer to the *FunctionExpression* and *ArrowFunctionExpression* forms respectively. When referring to either, the generic term ***function expression*** is used.

The type of a function expression is an object type containing a single call signature with parameter and return types inferred from the function expression's signature and body.

The descriptions of function declarations provided in section 6.1 apply to function expressions as well, except that function expressions do not support overloading.

4.9.1 Standard Function Expressions

Standard function expressions are function expressions written with the `function` keyword. The type of `this` in a standard function expression is the Any type.

Standard function expressions are transformed to JavaScript in the same manner as function declarations (see section 6.5).

4.9.2 Arrow Function Expressions

TypeScript supports ***arrow function expressions***, a new feature planned for ECMAScript 6. Arrow function expressions are a compact form of function expressions that omit the `function` keyword and have lexical scoping of `this`.

An arrow function expression of the form

ArrowFormalParameters `=>` *AssignmentExpression*

is exactly equivalent to

ArrowFormalParameters `=> { return` *AssignmentExpression* `; }`

Furthermore, arrow function expressions of the forms

Identifier `=>` *Block*
Identifier `=>` *AssignmentExpression*

are exactly equivalent to

`(` *Identifier* `) =>` *Block*
`(` *Identifier* `) =>` *AssignmentExpression*

Thus, the following examples are all equivalent:

```
(x) => { return Math.sin(x); }
(x) => Math.sin(x)
x => { return Math.sin(x); }
x => Math.sin(x)
```

A function expression using the `function` keyword introduces a new dynamically bound `this`, whereas an arrow function expression preserves the `this` of its enclosing context. Arrow function expressions are particularly useful for writing callbacks, which otherwise often have an undefined or unexpected `this`.

In the example

```
class Messenger {
    message = "Hello World";
    start() {
        setTimeout(() => alert(this.message), 3000);
    }
};
var messenger = new Messenger();
messenger.start();
```

the use of an arrow function expression causes the callback to have the same `this` as the surrounding 'start' method. Writing the callback as a standard function expression it becomes necessary to manually arrange access to the surrounding `this`, for example by copying it into a local variable:

```
class Messenger {
    message = "Hello World";
    start() {
        var _this = this;
        setTimeout(function() { alert(_this.message); }, 3000);
    }
};
var messenger = new Messenger();
messenger.start();
```

The TypeScript compiler applies this type of transformation to rewrite arrow function expressions into standard function expressions.

A construct of the form

```
< Identifier > ( ParamList ) => { ... }
```

could be parsed as an arrow function expression with a type parameter or a type assertion applied to an arrow function with no type parameter. It is resolved as the former, but parentheses can be used to select the latter meaning:

```
< Identifier > ( ( ParamList ) => { ... } )
```

4.9.3 Contextually Typed Function Expressions

Function expressions with no type parameters and no parameter or return type annotations (but possibly with optional parameters and default parameter values) are contextually typed in certain circumstances, as described in section 4.18. When a function expression is contextually typed by a function type *T*, the function expression is processed as if it had explicitly specified parameter type annotations as they exist in *T*. Parameters are matched by position and need not have matching names. If the function expression has fewer parameters than *T*, the additional parameters in *T* are ignored. If the function expression has more parameters than *T*, the additional parameters are all considered to have type Any.

When a function expression is contextually typed by a generic function type, parameters may be given types that are not directly denotable. In the example

```
interface Comparable<T> {
    compareTo(other: T): number;
}

interface Comparer {
    <T extends Comparable<T>>(x: T, y: T): number;
}

var f: Comparer = (x, y) => {
    var z: any;  // No name for type of x and y
    ...
}
```

the 'Comparer' type represents a function that can compare two values of some type 'T', where 'T' must implement 'Comparable<T>'. In the function expression assigned to 'f', contextual typing provides a type for 'x' and 'y' that is identical in structure to the type parameter in 'Comparer' (i.e. 'x' and 'y' will have a 'compareTo' method taking an argument of the same type as themselves), but no name is introduced for the type. In order to introduce a name for the type the function expression must be explicitly typed:

```
var f: Comparer = <T extends Comparable<T>>(x: T, y: T) => {
    var z: T;  // Z has same type as x and y
    ...
}
```

4.10 Property Access

A property access uses either dot notation or bracket notation. A property access expression is always classified as a reference.

A property access applied to a value of the primitive type Number, Boolean, or String behaves exactly as a property access applied to an object of the global interface type 'Number', 'Boolean', or 'String' respectively. Likewise, a property access applied to a value of an enum type behaves exactly as a property access applied to an object of the global interface type 'Number'.

In a property access, an object (including Number, Boolean, or String) appears to have additional properties that originate in the 'Object' or 'Function' global interface types, as described in section 3.3, and a type parameter appears to have the properties of its declared constraint, as described in section 3.4.

A dot notation property access of the form

ObjExpr **.** *Name*

where *ObjExpr* is an expression and *Name* is an identifier (including, possibly, a reserved word), is used to access the property with the given name on the given object. A dot notation property access is processed as follows at compile-time:

- If *ObjExpr* is of type Any, any *Name* is permitted and the property access is of type Any.

- Otherwise, if *Name* denotes a property member in the type of *ObjExpr*, the property access is of the type of that property.
- Otherwise, the property access is invalid and a compile-time error occurs.

A bracket notation property access of the form

ObjExpr [*IndexExpr*]

where *ObjExpr* and *IndexExpr* are expressions, is used to access the property with the name computed by the index expression on the given object. A bracket notation property access is processed as follows at compile-time:

- If *IndexExpr* is a string literal or a numeric literal and *ObjExpr* is of a type that has a property with the name given by that literal (converted to its string representation in the case of a numeric literal), the property access is of the type of that property.
- Otherwise, if *ObjExpr* is of a type that has a numeric index signature and *IndexExpr* is of type Any, the Number primitive type, or an enum type, the property access is of the type of that index signature.
- Otherwise, if *ObjExpr* is of a type that has a string index signature and *IndexExpr* is of type Any, the String or Number primitive type, or an enum type, the property access is of the type of that index signature.
- Otherwise, if *IndexExpr* is of type Any, the String or Number primitive type, or an enum type, the property access is of type Any.
- Otherwise, the property access is invalid and a compile-time error occurs.

4.11 The new Operator

A `new` operation has one of the following forms:

`new` *ConstructExpr*

`new` *ConstructExpr* (*Args*)

where *ConstructExpr* is an expression and *Args* is an argument list. The first form is equivalent to supplying an empty argument list. *ConstructExpr* must be of type Any or of an object type with one or more construct or call signatures. The operation is processed as follows at compile-time:

- If *ConstructExpr* is of type Any, *Args* can be any argument list and the result of the operation is of type Any.
- If *ConstructExpr* is of an object type with one or more construct signatures, the expression is processed in the same manner as a function call, but using the construct signatures as the initial set of candidate signatures for overload resolution. The result type of the function call becomes the result type of the operation.

- If *ConstructExpr* is of an object type with no construct signatures but one or more call signatures, the expression is processed as a function call. A compile-time error occurs if the result of the function call is not Void. The type of the result of the operation is Any.

4.12 Function Calls

Function calls are extended from JavaScript to optionally include type arguments.

Arguments: (Modified)
TypeArguments$_{opt}$ (*ArgumentList*$_{opt}$)

A function call takes one of the forms

FuncExpr (*Args*)

FuncExpr < *TypeArgs* > (*Args*)

where *FuncExpr* is an expression of a function type or of type Any, *TypeArgs* is a type argument list, and *Args* is an argument list.

If *FuncExpr* is of type Any, or of an object type that has no call signatures but is a subtype of the Function interface, the call is an ***untyped function call***. In an untyped function call no *TypeArgs* are permitted, *Args* can be any argument list, no contextual types are provided for the argument expressions, and the result is always of type Any.

If *FuncExpr* is of a function type, the call is a ***typed function call***. TypeScript employs ***overload resolution*** in typed function calls in order to support functions with multiple call signatures. Furthermore, TypeScript may perform ***type argument inference*** to automatically determine type arguments in generic function calls.

4.12.1 Overload Resolution

The purpose of overload resolution in a function call is to ensure that at least one signature is applicable, to provide contextual types for the arguments, and to determine the result type of the function call, which could differ between the multiple applicable signatures. Overload resolution has no impact on the run-time behavior of a function call. Since JavaScript doesn't support function overloading, all that matters at run-time is the name of the function.

The compile-time processing of a typed function call consists of the following steps:

- First, a set of candidate signatures is constructed from the signatures in the function type.
    - A non-generic signature is a candidate when
        - the function call has no type arguments, and
        - the signature is applicable with respect to the argument list of the function call.
    - A generic signature is a candidate in a function call without type arguments when
        - type inference (section 4.12.2) succeeds in inferring a list of type arguments,

        - the inferred type arguments satisfy their constraints, and
        - once the inferred type arguments are substituted for their associated type parameters, the signature is applicable with respect to the argument list of the function call.
    - A generic signature is a candidate in a function call with type arguments when
        - The signature has the same number of type parameters as were supplied in the type argument list,
        - the type arguments satisfy their constraints, and
        - once the type arguments are substituted for their associated type parameters, the signature is applicable with respect to the argument list of the function call.
- If the set of candidate signatures is empty, the function call is an error.
- Otherwise, for every signature *S* in the set, if another signature is a better match for the argument list, *S* is eliminated from the set.
- The result type of the call is the return type of the first remaining signature in the set in declaration order. For classes and interfaces, inherited signatures are considered to follow explicitly declared signatures in `extends` clause order.

A signature is said to be an ***applicable signature*** with respect to an argument list when

- each non-optional parameter has a corresponding argument, and
- each argument expression, contextually typed (section 4.18) by the corresponding parameter type in the signature, is of a type that is assignable to (section 3.8.3) that parameter type.

Given an argument list with a set of argument expressions { e_1, e_2, ..., e_n } and two applicable signatures *P* and *Q* with parameter types { P_1, P_2, ..., P_n } and { Q_1, Q_2, ..., Q_n }, *P* is a ***better match*** than *Q* when

- for no argument, e_X better matches Q_X than P_X, and
- for at least one argument, e_X better matches P_X than Q_X.

Given an expression *e* and two types T_1 and T_2, the better matching type is determined as follows:

- If T_1 and T_2 are identical types, neither is a better match.
- If *e* is a string literal and T_1 is the corresponding string literal type, T_1 is the better match.
- If *e* is a string literal and T_2 is the corresponding string literal type, T_2 is the better match.
- If the type of *e* is identical to T_1, T_1 is the better conversion.
- If the type of *e* is identical to T_2, T_2 is the better conversion.
- If T_1 is a subtype of T_2, T_1 is the better match.
- If T_2 is a subtype of T_1, T_2 is the better match.
- Otherwise, neither type is a better match.

4.12.2 Type Argument Inference

Given a signature $< T_1 , T_2 , \ldots , T_n >$ ($p_1 : P_1 , p_2 : P_2 , \ldots , p_m : P_m$), where each parameter type *P* references zero or more of the type parameters *T*, and an argument list ($e_1 , e_2 , \ldots , e_m$), the task of type argument

inference is to find a set of type arguments $A_1 \ldots A_n$ to substitute for $T_1 \ldots T_n$ such that the argument list becomes an applicable signature.

The ***inferred type argument*** for a particular type parameter is the best common type (section 3.10) of a set of candidate types. In order to compute candidate types, the argument list is processed as follows:

- Initially all inferred type arguments are considered ***unfixed*** with an empty set of candidate types.
- Proceeding from left to right, each argument expression *e* is ***inferentially typed*** by its corresponding parameter type *P*, possibly causing some inferred type arguments to become ***fixed***, and candidate type inferences are made for unfixed inferred type arguments by ***relating*** the type computed for *e* to *P*.

The process of inferentially typing an expression *e* by a type *T* is the same as that of contextually typing *e* by *T*, with the following exceptions:

- Where expressions contained within *e* would be contextually typed, they are instead inferentially typed.
- Where a contextual type would be included in a candidate set for a best common type (such as when inferentially typing an object or array literal), an inferential type is not.
- When a function expression is inferentially typed (section 4.9.3) and a type assigned to a parameter in that function expression references type parameters for which inferences are being made, the corresponding inferred type arguments to become ***fixed*** and no further candidate inferences are made for them.

Candidate inferences are made by relating a type *S* to a type *T* as follows:

- If *S* is not assignable to *T* when type Any is substituted for all type parameters, no candidate inferences are made.
- Otherwise, if *T* is a type parameter with an unfixed inferred type argument, *S* is added to the set of candidate types for that type argument.
- Otherwise, if *S* and *T* are object types, then for each member *M* in *T*:
    - If *M* is a property and *S* contains a property *N* with the same name as *M*, inferences are made by relating the type of *N* to the type of *M*.
    - If *M* is a call or construct signature, then for each signature *N* in *S* that is assignable to *M* when type Any is substituted for all type parameters, candidate inferences are made by relating parameter types in *N* to parameter types in the same position in *M*, and by relating the return type of *N* to the return type of *M*.
    - If *M* is a string index signature, then for each string index signature *N* in *S*, inferences are made by relating the type of *N* to the type of *M*.
    - If *M* is a numeric index signature, then for each string or numeric index signature *N* in *S*, inferences are made by relating the type of *N* to the type of *M*.

TODO: Examples. Include example that demonstrates the inferred type is Any when the set of candidate inferences is empty.

4.12.3 Grammar Ambiguities

The inclusion of type arguments in the *Arguments* production (section 4.12) gives rise to certain ambiguities in the grammar for expressions. For example, the statement

```
f(g<A, B>(7));
```

could be interpreted as a call to 'f' with two arguments, 'g < A' and 'B > (7)'. Alternatively, it could be interpreted as a call to 'f' with one argument, which is a call to a generic function 'g' with two type arguments and one regular argument.

The grammar ambiguity is resolved as follows: In a context where one possible interpretation of a sequence of tokens is an *Arguments* production, if the initial sequence of tokens forms a syntactically correct *TypeArguments* production and is followed by a '(' token, then the sequence of tokens is processed an *Arguments* production, and any other possible interpretation is discarded. Otherwise, the sequence of tokens is not considered an *Arguments* production.

This rule means that the call to 'f' above is interpreted as a call with one argument, which is a call to a generic function 'g' with two type arguments and one regular argument. However, the statements

```
f(g < A, B > 7);
```

```
f(g < A, B > +(7));
```

are both interpreted as calls to 'f' with two arguments.

4.13 Type Assertions

TypeScript extends the JavaScript expression grammar with the ability to assert a type for an expression:

UnaryExpression: (Modified)
    ...
    < *Type* > *UnaryExpression*

A type assertion expression consists of a type enclosed in < and > followed by a unary expression. Type assertion expressions are purely a compile-time construct. Type assertions are *not* checked at run-time and have no impact on the emitted JavaScript (and therefore no run-time cost). The type and the enclosing < and > are simply removed from the generated code.

A type assertion expression of the form < *T* > *e* requires the type of *e* to be assignable to *T* or *T* to be assignable to the type of *e*, or otherwise a compile-time error occurs. The type of the result is *T*.

Type assertions check for assignment compatibility in both directions. Thus, type assertions allow type conversions that *might* be correct, but aren't *known* to be correct. In the example

```
class Shape { ... }

class Circle extends Shape { ... }
```

```
function createShape(kind: string): Shape {
    if (kind === "circle") return new Circle();
    ...
}

var circle = <Circle> createShape("circle");
```

the type annotations indicate that the 'createShape' function *might* return a 'Circle' (because 'Circle' is a subtype of 'Shape'), but isn't *known* to do so (because its return type is 'Shape'). Therefore, a type assertion is needed to treat the result as a 'Circle'.

As mentioned above, type assertions are not checked at run-time and it is up to the programmer to guard against errors, for example using the `instanceof` operator:

```
var shape = createShape(shapeKind);
if (shape instanceof Circle) {
    var circle = <Circle> shape;
    ...
}
```

4.14 Unary Operators

The subsections that follow specify the compile-time processing rules of the unary operators. In general, if the operand of a unary operator does not meet the stated requirements, a compile-time error occurs and the result of the operation defaults to type Any in further processing.

4.14.1 The ++ and -- operators

These operators, in prefix or postfix form, require their operand to be of type Any, the Number primitive type, or an enum type, and classified as a reference (section 4.1). They produce a result of the Number primitive type.

4.14.2 The +, –, and ~ operators

These operators permit their operand to be of any type and produce a result of the Number primitive type.

The unary + operator can conveniently be used to convert a value of any type to the Number primitive type:

```
function getValue() { ... }

var n = +getValue();
```

The example above converts the result of 'getValue()' to a number if it isn't a number already. The type inferred for 'n' is the Number primitive type regardless of the return type of 'getValue'.

4.14.3 The ! operator

The ! operator permits its operand to be of any type and produces a result of the Boolean primitive type.

Two unary ! operators in sequence can conveniently be used to convert a value of any type to the Boolean primitive type:

```
function getValue() { ... }

var b = !!getValue();
```

The example above converts the result of 'getValue()' to a Boolean if it isn't a Boolean already. The type inferred for 'b' is the Boolean primitive type regardless of the return type of 'getValue'.

4.14.4 The delete Operator

The `delete` operator takes an operand of any type and produces a result of the Boolean primitive type.

4.14.5 The void Operator

The `void` operator takes an operand of any type and produces the value `undefined`. The type of the result is the Undefined type (3.2.6).

4.14.6 The typeof Operator

The `typeof` operator takes an operand of any type and produces a value of the String primitive type.

4.15 Binary Operators

The subsections that follow specify the compile-time processing rules of the binary operators. In general, if the operands of a binary operator do not meet the stated requirements, a compile-time error occurs and the result of the operation defaults to type any in further processing. Tables that summarize the compile-time processing rules for operands of the Any type, the Boolean, Number, and String primitive types, and all object types and type parameters (the Object column in the tables) are provided.

4.15.1 The *, /, %, –, <<, >>, >>>, &, ^, and | operators

These operators require their operands to be of type Any, the Number primitive type, or an enum type. Operands of an enum type are treated as having the primitive type Number. If one operand is the `null` or `undefined` value, it is treated as having the type of the other operand. The result is always of the Number primitive type.

	Any	Boolean	Number	String	Object
Any	Number		Number		
Boolean					
Number	Number		Number		
String					
Object					

4.15.2 The + operator

The binary + operator requires both operands to be of the Number primitive type or an enum type, or at least one of the operands to be of type Any or the String primitive type. Operands of an enum type are treated as having the primitive type Number. If one operand is the `null` or `undefined` value, it is treated as having the type of the other operand. If both operands are of the Number primitive type, the result is of the Number primitive type. If one or both operands are of the String primitive type, the result is of the String primitive type. Otherwise, the result is of type Any.

	Any	Boolean	Number	String	Object
Any	Any	Any	Any	String	Any
Boolean	Any			String	
Number	Any		Number	String	
String	String	String	String	String	String
Object	Any			String	

A value of any type can converted to the String primitive type by adding an empty string:

```
function getValue() { ... }

var s = getValue() + "";
```

The example above converts the result of 'getValue()' to a string if it isn't a string already. The type inferred for 's' is the String primitive type regardless of the return type of 'getValue'.

4.15.3 The <, >, <=, >=, ==, !=, ===, and !== operators

These operators require one operand type to be identical to or a subtype of the other operand type. The result is always of the Boolean primitive type.

	Any	Boolean	Number	String	Object
Any	Boolean	Boolean	Boolean	Boolean	Boolean
Boolean	Boolean	Boolean			
Number	Boolean		Boolean		
String	Boolean			Boolean	
Object	Boolean				Boolean

4.15.4 The instanceof operator

The `instanceof` operator requires the left operand to be of type Any, an object type, or a type parameter type, and the right operand to be of type Any or a subtype of the 'Function' interface type. The result is always of the Boolean primitive type.

Note that object types containing one or more call or construct signatures are automatically subtypes of the 'Function' interface type, as described in section 3.3.

4.15.5 The in operator

The `in` operator requires the left operand to be of type Any or the String primitive type, and the right operand to be of type Any, an object type, or a type parameter type. The result is always of the Boolean primitive type.

4.15.6 The && operator

The && operator permits the operands to be of any type and produces a result of the same type as the second operand.

	Any	Boolean	Number	String	Object
Any	Any	Boolean	Number	String	Object
Boolean	Any	Boolean	Number	String	Object
Number	Any	Boolean	Number	String	Object
String	Any	Boolean	Number	String	Object
Object	Any	Boolean	Number	String	Object

4.15.7 The || operator

The || operator permits the operands to be of any type and produces a result that is of the best common type (section 3.10) of the two operand types.

	Any	Boolean	Number	String	Object
Any	Any	Any	Any	Any	Any
Boolean	Any	Boolean	{ }	{ }	{ }
Number	Any	{ }	Number	{ }	{ }
String	Any	{ }	{ }	String	{ }
Object	Any	{ }	{ }	{ }	Object

4.16 The Conditional Operator

In a conditional expression of the form

Cond ? *Expr1* : *Expr2*

the *Cond* expression may be of any type, and *Expr1* and *Expr2* expressions must be of identical types or the type of one must be a subtype of the other. The result is of the best common type (section 3.10) of the two expression types.

4.17 Assignment Operators

An assignment of the form

VarExpr = *ValueExpr*

requires *VarExpr* to be classified as a reference (section 4.1). *ValueExpr* is contextually typed (section 4.18) by the type of *VarExpr*, and the type of *ValueExpr* must be assignable to (section 3.8.3) the type of *VarExpr*, or otherwise a compile-time error occurs. The result is a value with the type of *ValueExpr*.

A compound assignment of the form

VarExpr *Operator*= *ValueExpr*

where *Operator*= is of the compound assignment operators

*= /= %= += -= <<= >>= >>>= &= ^= |=

is subject to the same requirements, and produces a value of the same type, as the corresponding non-compound operation. A compound assignment furthermore requires *VarExpr* to be classified as a reference (section 4.1) and the type of the non-compound operation to be assignable to the type of *VarExpr*.

4.18 Contextually Typed Expressions

In certain situations, parameter and return types of function expressions are automatically inferred from the contexts in which the function expressions occur. For example, given the declaration

```
var f: (s: string) => string;
```

the assignment

```
f = function(s) { return s.toLowerCase(); }
```

infers the type of the 's' parameter to be the String primitive type even though there is no type annotation to that effect. The function expression is said to be ***contextually typed*** by the variable to which it is being assigned. Contextual typing occurs in the following situations:

- In variable and member declarations with a type annotation and an initializer, the initializer expression is contextually typed by the type of the variable or property.
- In assignment expressions, the right hand expression is contextually typed by the type of the left hand expression.
- In typed function calls, argument expressions are contextually typed by their parameter types.
- In return statements, if the containing function has a known return type, the expression is contextually typed by that return type. A function's return type is known if the function includes a return type annotation or is itself contextually typed.
- In contextually typed object literals, property assignments are contextually typed by their property types.
- In contextually typed array literals, element expressions are contextually typed by the array element type.

Contextual typing of an expression *e* by a type *T* proceeds as follows:

- If *e* is an *ObjectLiteral* and *T* is an object type, *e* is processed with the contextual type *T*, as described in section 4.5.
- If *e* is an *ArrayLiteral* and *T* is an object type with a numeric index signature, *e* is processed with the contextual type *T*, as described in section 4.6.
- If *e* is a *FunctionExpression* or *ArrowFunctionExpression* with no type parameters and no parameter or return type annotations, and *T* is an object type with exactly one call signature, *e* is processed with the contextual type *T*, as described in section 4.9.3.
- Otherwise, *e* is processed without a contextual type.

The rules above require expressions be of the exact syntactic forms specified in order to be processed as contextually typed constructs. For example, given the declaration of the variable 'f' above, the assignment

```
f = s => s.toLowerCase();
```

causes the function expression to be contextually typed, inferring the String primitive type for 's'. However, simply enclosing the construct in parentheses

```
f = (s => s.toLowerCase());
```

causes the function expression to be processed without a contextual type, now inferring ‘s’ and the result of the function to be of type Any as no type annotations are present.

In the following example

```
interface EventObject {
    x: number;
    y: number;
}

interface EventHandlers {
    mousedown?: (event: EventObject) => void;
    mouseup?: (event: EventObject) => void;
    mousemove?: (event: EventObject) => void;
}

function setEventHandlers(handlers: EventHandlers) { ... }

setEventHandlers({
    mousedown: e => { startTracking(e.x, e.y); },
    mouseup: e => { endTracking(); }
});
```

the object literal passed to ‘setEventHandlers’ is contextually typed to the ‘EventHandlers’ type. This causes the two property assignments to be contextually typed to the unnamed function type ‘(event: EventObject) => void’, which in turn causes the ‘e’ parameters in the arrow function expressions to automatically be typed as ‘EventObject’.

5 Statements

TODO: Describe type checking for statements.

5.1 Variable Statements

Variable statements are extended to include optional type annotations.

VariableDeclaration: (Modified)
    Identifier *TypeAnnotation*$_{opt}$ *Initialiser*$_{opt}$

VariableDeclarationNoIn: (Modified)
    Identifier *TypeAnnotation*$_{opt}$ *InitialiserNoIn*$_{opt}$

TypeAnnotation:
    `:` *Type*

A variable declaration introduces a variable with the given name in the containing declaration space. The type associated with a variable is determined as follows:

- If the declaration includes a type annotation, the stated type becomes the type of the variable. If an initializer is present, the initializer expression is contextually typed (see section 4.18) by the stated type and must be assignable to the stated type, or otherwise a compile-time error occurs.
- If the declaration includes an initializer but no type annotation, the widened type (see section 3.9) of the initializer expression becomes the type of the property.
- If the declaration includes neither a type annotation nor an initializer, the type of the variable becomes the Any type.

Multiple declarations for the same variable name in the same declaration space are permitted, provided that each declaration associates the same type with the variable.

Below are some examples of variable declarations and their associated types.

```
var a;                          // any
var b: number;                  // number
var c = 1;                      // number
var d = { x: 1, y: "hello" };   // { x: number; y: string; }
var e: any = "test";            // any
```

The following is permitted because all declarations of the single variable ‘x’ associate the same type (Number) with ‘x’.

```
var x = 1;
var x: number;
if (x == 1) {
    var x = 2;
}
```

In the following example, all five variables are of the same type, '{ x: number; y: number; }'.

```
interface Point { x: number; y: number; }

var a = { x: 0, y: <number> undefined };
var b: Point = { x: 0; y: undefined };
var c = <Point> { x: 0, y: undefined };
var d: { x: number; y: number; } = { x: 0, y: undefined };
var e = <{ x: number; y: number; }> { x: 0, y: undefined };
```

6 Functions

TypeScript extends JavaScript functions to include type parameters, parameter and return type annotations, overloads, default parameter values, and rest parameters.

6.1 Function Declarations

Function declarations consist of an optional set of function overloads followed by an actual function implementation.

FunctionDeclaration: *(Modified)*
    FunctionOverloads$_{opt}$ *FunctionImplementation*

FunctionOverloads:
    FunctionOverload
    FunctionOverloads *FunctionOverload*

FunctionOverload:
    `function` *Identifier* *CallSignature* `;`

FunctionImplementation:
    `function` *Identifier* *CallSignature* `{` *FunctionBody* `}`

A function declaration introduces a function with the given name in the containing declaration space. Function overloads, if present, must specify the same name as the function implementation. If a function declaration includes overloads, the overloads determine the call signatures of the type given to the function object and the function implementation signature must be assignable to that type. Otherwise, the function implementation itself determines the call signature. Function overloads have no other effect on a function declaration.

6.2 Function Overloads

Function overloads allow a more accurate specification of the patterns of invocation supported by a function than is possible with a single signature. The compile-time processing of a call to an overloaded function chooses the best candidate overload for the particular arguments and the return type of that overload becomes the result type the function call expression. Thus, using overloads it is possible to statically describe the manner in which a function's return type varies based on its arguments. Overload resolution in function calls is described further in section 4.12.

Function overloads are purely a compile-time construct. They have no impact on the emitted JavaScript and thus no run-time cost.

The parameter list of a function overload cannot specify default values for parameters. In other words, an overload may use only the `?` form when specifying optional parameters.

It is an error to declare multiple function overloads that are considered identical (section 3.8.1) or differ only in their return types.

The following is an example of a function with overloads.

```
function attr(name: string): string;
function attr(name: string, value: string): Accessor;
function attr(map: any): Accessor;
function attr(nameOrMap: any, value?: string): any {
    if (nameOrMap && typeof nameOrMap === "object") {
        // handle map case
    }
    else {
        // handle string case
    }
}
```

Note that each overload and the final implementation specify the same identifier. The type of the local variable 'attr' introduced by this declaration is

```
var attr: {
    (name: string): string;
    (name: string, value: string): Accessor;
    (map: any): Accessor;
};
```

Note that the signature of the actual function implementation is not included in the type.

6.3 Function Implementations

A function implementation without a return type annotation is said to be an ***implicitly typed function***. The return type of an implicitly typed function *f* is inferred from its function body as follows:

- If there are no return statements with expressions in *f*'s function body, the inferred return type is Void.
- Otherwise, if *f*'s function body directly references *f* or references any implicitly typed functions that through this same analysis reference *f*, the inferred return type is Any.
- Otherwise, the inferred return type is the widened form (section 3.9) of the best common type (section 3.10) of the types of the return statement expression in the function body, ignoring return statements with no expressions. A compile-time error occurs if the best common type isn't one of the return statement expression types (i.e. if the best common type is an empty type).

In the example

```
function f(x: number) {
    if (x <= 0) return x;
    return g(x);
}
```

```
function g(x: number) {
    return f(x - 1);
}
```

the inferred return type for 'f' and 'g' is Any because the functions reference themselves through a cycle with no return type annotations. Adding an explicit return type 'number' to either breaks the cycle and causes the return type 'number' to be inferred for the other.

The type of `this` in a function implementation is the Any type.

In the signature of a function implementation, a parameter can be marked optional by following it with an initializer. An optional parameter with an initializer but no type annotation has its type inferred from the initializer. Specifically, the type of such a parameter is the widened form of the type of the initializer expression. An initializer expression for a given parameter is permitted to reference parameters that are declared to the left of that parameter, but it is a compile-time error to access other parameters or locals. For each parameter with an initializer, a statement that substitutes the default value for an omitted argument is included in the generated JavaScript, as described in section 6.3. The example

```
function strange(x: number, y = x * 2, z = x + y) {
    return z;
}
```

generates JavaScript that is equivalent to

```
function strange(x, y, z) {
    if (typeof y === "undefined") { y = x * 2; }
    if (typeof z === "undefined") { z = x + y; }
    return z;
}
```

6.4 Generic Functions

A function implementation may include type parameters in its signature (section 3.7.2.1) and is then called a ***generic function***. Type parameters provide a mechanism for expressing relationships between parameter and return types in call operations. Type parameters have no run-time representation—they are purely a compile-time construct.

Type parameters declared in the signature of a function implementation are in scope in the signature and body of that function implementation.

The following is an example of a generic function:

```
interface Comparable<T> {
    localeCompare(other: T): number;
}
```

```
function compare<T extends Comparable<T>>(x: T, y: T): number {
    if (x == null) return y == null ? 0 : -1;
    if (y == null) return 1;
    return x.localeCompare(y);
}
```

Note that the 'x' and 'y' parameters are known to be subtypes of the constraint 'Comparable<T>' and therefore have a 'compareTo' member. This is described further in section 3.4.

The type arguments of a call to a generic function may be explicitly specified in a call operation or may, when possible, be inferred (section 4.12.2) from the types of the regular arguments in the call. In the example

```
class Person {
    name: string;
    localeCompare(other: Person) {
        return compare(this.name, other.name);
    }
}
```

the type argument to 'compare' is automatically inferred to be the String type because the two arguments are strings.

6.5 Code Generation

A function declaration generates JavaScript code that is equivalent to:

```
function <FunctionName>(<FunctionParameters>) {
    <DefaultValueAssignments>
    <FunctionStatements>
}
```

FunctionName is the name of the function (or nothing in the case of a function expression).

FunctionParameters is a comma separated list of the function's parameter names.

DefaultValueAssignments is a sequence of default property value assignments, one for each parameter with a default value, in the order they are declared, of the form

```
if (typeof <Parameter> === "undefined") { <Parameter> = <Default>; }
```

where *Parameter* is the parameter name and *Default* is the default value expression.

FunctionStatements is the code generated for the statements specified in the function body.

7 Interfaces

Interfaces provide the ability to name and parameterize object types and to compose existing named object types into new ones.

Interfaces have no run-time representation—they are purely a compile-time construct. Interfaces are particularly useful for documenting and validating the required shape of properties, objects passed as parameters, and objects returned from functions.

Because TypeScript has a structural type system, an interface type with a particular set of members is considered identical to, and can be substituted for, another interface type or object type literal with an identical set of members (see section 3.8.1).

Class declarations may reference interfaces in their implements clause to validate that they provide an implementation of the interfaces.

7.1 Interface Declarations

An interface declaration declares a new named type (section 3.5) by introducing a type name in the containing module.

InterfaceDeclaration:
`interface` *Identifier* *TypeParameters*$_{opt}$ *InterfaceExtendsClause*$_{opt}$ *ObjectType*

InterfaceExtendsClause:
`extends` *ClassOrInterfaceTypeList*

ClassOrInterfaceTypeList:
ClassOrInterfaceType
ClassOrInterfaceTypeList `,` *ClassOrInterfaceType*

ClassOrInterfaceType:
TypeReference

The *Identifier* of an interface declaration may not be one of the predefined type names (section 3.6.1).

An interface may optionally have type parameters (section 3.5.1) that serve as placeholders for actual types to be provided when the interface is referenced in type references. An interface with type parameters is called a ***generic interface***. The type parameters of a generic interface declaration are in scope in the entire declaration and may be referenced in the *InterfaceExtendsClause* and *ObjectType* body.

An interface can inherit from zero or more ***base types*** which are specified in the *InterfaceExtendsClause*. The base types must be type references to class or interface types.

An interface has the members specified in the *ObjectType* of its declaration and furthermore inherits all base type members that aren't hidden by declarations in the interface:

- A property declaration hides a public base type property with the same name.
- A call signature declaration hides a base type call signature that is identical when return types are ignored.
- A construct signature declaration hides a base type construct signature that is identical when return types are ignored.
- A string index signature declaration hides a base type string index signature.
- A numeric index signature declaration hides a base type numeric index signature.

The following constraints must be satisfied by an interface declaration or otherwise a compile-time error occurs:

- An interface declaration may not, directly or indirectly, specify a base type that originates in the same declaration. In other words an interface cannot, directly or indirectly, be a base type of itself, regardless of type arguments.
- An interface cannot declare a property with the same name as an inherited private property.
- Inherited properties with the same name must be identical (section 3.8.1).
- The instance type (section 3.5.3) of the declared interface must be a subtype (section 3.8.2) of each of the base type references.

An interface is permitted to inherit identical members from multiple base types and will in that case only contain one occurrence of each particular member.

Below is an example of two interfaces that contain properties with the same name but different types:

```
interface Mover {
    move(): void;
    getStatus(): { speed: number; };
}

interface Shaker {
    shake(): void;
    getStatus(): { frequency: number; };
}
```

An interface that extends 'Mover' and 'Shaker' must declare a new 'getStatus' property as it would otherwise inherit two 'getStatus' properties with different types. The new 'getStatus' property must be declared such that the resulting 'MoverShaker' is a subtype of both 'Mover' and 'Shaker':

```
interface MoverShaker extends Mover, Shaker {
    getStatus(): { speed: number; frequency: number; };
}
```

Since function and constructor types are just object types containing call and construct signatures, interfaces can be used to declare named function and constructor types. For example:

```
interface StringComparer { (a: string, b: string): number; }
```

This declares type 'StringComparer' to be a function type taking two strings and returning a number.

7.2 Declaration Merging

Interfaces are "open-ended" and interface declarations with the same qualified name relative to a common root (as defined in section 2.3) contribute to a single interface.

When a generic interface has multiple declarations, all declarations must have identical type parameter lists, i.e. identical type parameter names with identical constraints in identical order.

In an interface with multiple declarations, the `extends` clauses are merged into a single set of base types and the bodies of the interface declarations are merged into a single object type.

7.3 Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private members of a base class. When a class containing private members is the base type of an interface type, that interface type can only be implemented by that class or a descendant class. For example:

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}

class Button extends Control {
    select() { }
}

class TextBox extends Control {
    select() { }
}

class Image extends Control {
}

class Location {
    select() { }
}
```

In the above example, 'SelectableControl' contains all of the members of 'Control', including the private 'state' property. Since 'state' is a private member it is only possible for descendants of 'Control' to

implement ‘SelectableControl’. This is because only descendants of ‘Control’ will have a ‘state’ private member that originates in the same declaration, which is a requirement for private members to be compatible (section 3.8).

Within the ‘Control’ class it is possible to access the ‘state’ private member through an instance of ‘SelectableControl’. Effectively, a ‘SelectableControl’ acts like a ‘Control’ that is known to have a ‘select’ method. The ‘Button’ and ‘TextBox’ classes are subtypes of ‘SelectableControl’ (because they both inherit from ‘Control’ and have a ‘select’ method), but the ‘Image’ and ‘Location’ classes are not.

7.4 Dynamic Type Checks

TypeScript does not provide a direct mechanism for dynamically testing whether an object implements a particular interface. Instead, TypeScript code can use the JavaScript technique of checking whether an appropriate set of members are present on the object. For example, given the declarations in section 7.1, the following is a dynamic check for the ‘MoverShaker’ interface:

```
var obj: any = getSomeObject();
if (obj && obj.move && obj.shake && obj.getStatus) {
    var moverShaker = <MoverShaker> obj;
    ...
}
```

If such a check is used often it can be abstracted into a function:

```
function asMoverShaker(obj: any): MoverShaker {
    return obj && obj.move && obj.shake && obj.getStatus ? obj : null;
}
```

8 Classes

TypeScript supports classes that are closely aligned with those proposed for ECMAScript 6, and includes extensions for instance and static member declarations and properties declared and initialized from constructor parameters.

NOTE: TypeScript currently doesn't support class expressions or nested class declarations from the ECMAScript 6 proposal.

8.1 Class Declarations

Class declarations introduce named types and provide implementations of those types. Classes support inheritance, allowing derived classes to extend and specialize base classes.

ClassDeclaration:
`class` *Identifier* *TypeParameters*$_{opt}$ *ClassHeritage* `{` *ClassBody* `}`

A *ClassDeclaration* declares a ***class type*** and a ***constructor function***, both with the name given by *Identifier*, in the containing module. The class type is created from the instance members declared in the class body and the instance members inherited from the base class. The constructor function is created from the constructor declaration, the static member declarations in the class body, and the static members inherited from the base class. The constructor function initializes and returns an instance of the class type.

The *Identifier* of a class declaration may not be one of the predefined type names (section 3.6.1).

A class may optionally have type parameters (section 3.5.1) that serve as placeholders for actual types to be provided when the class is referenced in type references. A class with type parameters is called a ***generic class***. The type parameters of a generic class declaration are in scope in the entire declaration and may be referenced in the *ClassHeritage* and *ClassBody*.

The following example introduces both a named type called 'Point' (the class type) and a member called 'Point' (the constructor function) in the containing module.

```
class Point {
    constructor(public x: number, public y: number) { }
    public length() { return Math.sqrt(this.x * this.x + this.y * this.y);
    static origin = new Point(0, 0);
}
```

The 'Point' type is exactly equivalent to

```
interface Point {
    x: number;
    y: number;
    length(): number;
}
```

The 'Point' member is a constructor function whose type corresponds to the declaration

```
var Point: {
    new(x: number, y: number): Point;
    origin: Point;
};
```

The context in which a class is referenced distinguishes between the class instance type and the constructor function. For example, in the assignment statement

```
var p: Point = new Point(10, 20);
```

the identifier 'Point' in the type annotation refers to the class instance type, whereas the identifier 'Point' in the `new` expression refers to the constructor function object.

8.1.1 Class Heritage Specification

The heritage specification of a class consists of optional `extends` and `implements` clauses. The `extends` clause specifies the base class of the class and the `implements` clause specifies a set of interfaces for which to validate the class provides an implementation.

ClassHeritage:
    ClassExtendsClause$_{opt}$ *ImplementsClause*$_{opt}$

ClassExtendsClause:
    `extends` *ClassType*

ClassType:
    TypeReference

ImplementsClause:
    `implements` *ClassOrInterfaceTypeList*

A class that includes an `extends` clause is called a ***derived class***, and the class specified in the `extends` clause is called the ***base class*** of the derived class. When a class heritage specification omits the `extends` clause, the class does not have a base class. However, as is the case with every object type, type references (section 3.3.1) to the class will appear to have the members of the global interface type named 'Object' unless those members are hidden by members with the same name in the class.

The following constraints must be satisfied by the class heritage specification or otherwise a compile-time error occurs:

- A class declaration may not, directly or indirectly, specify a base class that originates in the same declaration. In other words an class cannot, directly or indirectly, be a base class of itself, regardless of type arguments.
- The *TypeName* part of the base class type reference specified in the `extends` clause is required to be a reference to the class constructor function when evaluated as an expression.
- The instance type (section 3.5.3) of the declared class must be a subtype (section 3.8.2) of the base type reference and each of the type references listed in the `implements` clause.
- The constructor function type created by the class declaration must be a subtype of the base class constructor function type, ignoring construct signatures.

The following example illustrates a situation in which the second rule above would be violated:

```
class A { a: number; }

module Foo {
    var A = 1;
    class B extends A { b: string; }
}
```

When evaluated as an expression, the type reference 'A' in the `extends` clause doesn't reference the class constructor function of 'A' (instead it references the local variable 'A').

The only situation in which the last two constraints above are violated is when a class overrides one or more base class members with incompatible new members.

Note that because TypeScript has a structural type system, a class doesn't need to explicitly state that it implements an interface—it suffices for the class to simply contain the appropriate set of instance members. The `implements` clause of a class provides a mechanism to assert and validate that the class contains the appropriate sets of instance members, but otherwise it has no effect on the class type.

8.1.2 Class Body

The class body consists of zero or more constructor or member declarations. Statements are not allowed in the body of a class—they must be placed in the constructor or in members.

ClassBody:
    ClassElements$_{opt}$

ClassElements:
    ClassElement
    ClassElements *ClassElement*

ClassElement:
    ConstructorDeclaration
    MemberDeclaration
    IndexSignature

The body of class may optionally contain a single constructor declaration. Constructor declarations are described in section 8.3.

Member declarations are used to declare instance and static members of the class. Member declarations are described in section 8.4.

8.2 Members

The members of a class consist of the members introduced through member declarations in the class body and the members inherited from the base class.

8.2.1 Instance and Static Members

Members are either ***instance members*** or ***static members***.

Instance members are members of the class type (section 8.2.4) and its associated instance type. Within constructors, instance member functions, and instance member accessors, the type of `this` is the instance type (section 3.5.3) of the class.

Static members are declared using the `static` modifier and are members of the constructor function type (section 8.2.5). Within static member functions and static member accessors, the type of `this` is the constructor function type excluding construct signatures.

Type parameters cannot be referenced in static member declarations.

8.2.2 Accessibility

Members have either ***public*** or ***private*** accessibility. The default is public accessibility, but member declarations may include a `public` or `private` modifier to explicitly specify the desired accessibility.

Public members can be accessed everywhere, but private members can be accessed only within the class body that contains their declaration. Any attempt to access a private member outside the class body that contains its declaration results in a compile-time error.

Private accessibility is enforced only at compile-time and serves as no more than an *indication of intent*. Since JavaScript provides no mechanism to create private properties on an object, it is not possible to enforce the private modifier in dynamic code at run-time. For example, private accessibility can be defeated by changing an object's static type to Any and accessing the member dynamically.

8.2.3 Inheritance and Overriding

A derived class ***inherits*** all members from its base class it doesn't ***override***. Inheritance means that a derived class implicitly contains all non-overridden members of the base class. Both public and private members are inherited, but only public members can be overridden. A member in a derived class is said to override a member in a base class when the derived class member has the same name and kind (instance or static) as the base class member. The type of an overriding member must be a subtype (section 3.8.2) of the type of the overridden member, or otherwise a compile-time error occurs.

Base class instance member functions can be overridden by derived class instance member functions, but not by other kinds of members.

Base class instance member variables and accessors can be overridden by derived class instance member variables and accessors, but not by other kinds of members.

Base class static members can be overridden by derived class static members of any kind as long as the types are compatible, as described above.

8.2.4 Class Types

An class declaration declares a new named type (section 3.5) called a class type. Within the constructor and member functions of a class, the type of `this` is the instance type (section 3.5.3) of this class type. The class type has the following members:

- A property for each instance member variable declaration in the class body.
- A property of a function type for each instance member function declaration in the class body.
- A property for each uniquely named instance member accessor declaration in the class body.
- A property for each constructor parameter declared with a `public` or `private` modifier.
- All base class instance type properties that are not overridden in the class.

In the example

```
class A {
    public x: number;
    public f() { }
    public g(a: any) { return undefined; }
    static s: string;
}

class B extends A {
    public y: number;
    public g(b: boolean) { return false; }
}
```

the instance type of 'A' is

```
interface A {
    x: number;
    f: () => void;
    g: (a: any) => any;
}
```

and the instance type of 'B' is

```
interface B {
    x: number;
    y: number;
    f: () => void;
    g: (b: boolean) => boolean;
}
```

Note that static declarations in a class do not contribute to the class type and its instance type—rather, static declarations introduce properties on the constructor function object. Also note that the declaration of 'g' in 'B' overrides the member inherited from 'A'.

8.2.5 Constructor Function Types

The type of the constructor function introduced by a class declaration is called the constructor function type. The constructor function type has the following members:

- If the class contains no constructor declaration and has no base class, a single construct signature with no parameters, having the same type parameters as the class and returning the instance type of the class.
- If the class contains no constructor declaration and has a base class, a set of construct signatures with the same parameters as those of the base class constructor function type following substitution of type parameters with the type arguments specified in the base class type reference, all having the same type parameters as the class and returning the instance type of the class.
- If the class contains a constructor declaration with no overloads, a construct signature with the parameter list of the constructor implementation, having the same type parameters as the class and returning the instance type of the class.
- If the class contains a constructor declaration with overloads, a set of construct signatures with the parameter lists of the overloads, all having the same type parameters as the class and returning the instance type of the class.
- A property for each static member variable declaration in the class body.
- A property of a function type for each static member function declaration in the class body.
- A property for each uniquely named static member accessor declaration in the class body.
- A property named 'prototype' of the class instance type.
- All base class constructor function type properties that are not overridden in the class.

The example

```
class Pair<T1, T2> {
    constructor(public item1: T1, public item2: T2) { }
}

class TwoArrays<T> extends Pair<T[], T[]> { }
```

introduces two named types corresponding to

```
interface Pair<T1, T2> {
    item1: T1;
    item2: T2;
}

interface TwoArrays<T> {
    item1: T[];
    item2: T[];
}
```

and two constructor functions corresponding to

```
var Pair: {
    new <T1, T2>(item1: T1, item2: T2): Pair<T1, T2>;
}

var TwoArrays: {
    new <T>(item1: T[], item2: T[]): TwoArrays<T>;
}
```

Note that the construct signatures in the constructor function types have the same type parameters as their class and return the instance type of their class. Also note that when a derived class doesn't declare a constructor, type arguments from the base class reference are substituted before construct signatures are propagated from the base constructor function type to the derived constructor function type.

8.3 Constructor Declarations

A constructor declaration declares the constructor function of a class.

ConstructorDeclaration:
ConstructorOverloads$_{opt}$ *ConstructorImplementation*

ConstructorOverloads:
ConstructorOverload
ConstructorOverloads *ConstructorOverload*

ConstructorOverload:
`constructor` `(` *ParameterList*$_{opt}$ `)` `;`

ConstructorImplementation:
`constructor` `(` *ParameterList*$_{opt}$ `)` `{` *FunctionBody* `}`

A class may contain at most one constructor declaration. If a class contains no constructor declaration, an automatic constructor is provided, as described in section 8.3.3.

If a constructor declaration includes overloads, the overloads determine the construct signatures of the type given to the constructor function object, and the constructor implementation signature must be

assignable to that type. Otherwise, the constructor implementation itself determines the construct signature. This exactly parallels the way overloads are processed in a function declaration (section 6.2).

The function body of a constructor is permitted to contain return statements. If return statements specify expressions, those expressions must be of types that are assignable to the instance type of the class.

The type parameters of a generic class are in scope and accessible in a constructor declaration.

8.3.1 Constructor Parameters

Similar to functions, only the constructor implementation (and not constructor overloads) can specify default value expressions for optional parameters. It is a compile-time error for such default value expressions to reference `this`. For each parameter with a default value, a statement that substitutes the default value for an omitted argument is included in the JavaScript generated for the constructor function.

A parameter of a *ConstructorImplementation* may be prefixed with a `public` or `private` modifier. This is called a ***parameter property declaration*** and is shorthand for declaring a property with the same name as the parameter and initializing it with the value of the parameter. For example, the declaration

```
class Point {
    constructor(public x: number, public y: number) {
        // Constructor body
    }
}
```

is equivalent to writing

```
class Point {
    public x: number;
    public y: number;
    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
        // Constructor body
    }
}
```

8.3.2 Super Calls

Super calls (section 4.8.1) are used to call the constructor of the base class. A super call consists of the keyword `super` followed by an argument list enclosed in parentheses. For example:

```
class ColoredPoint extends Point {
    constructor(x: number, y: number, public color: string) {
        super(x, y);
    }
}
```

Constructors of classes with no `extends` clause may not contain super calls, whereas constructors of derived classes must contain at least one super call somewhere in their function body. Super calls are not permitted outside constructors or in local functions inside constructors.

The first statement in the body of a constructor *must* be a super call if both of the following are true:

- The containing class is a derived class.
- The constructor declares parameter properties or the containing class declares instance member variables with initializers.

In such a required super call, it is a compile-time error for argument expressions to reference `this`.

Initialization of parameter properties and instance member variables with initializers takes place immediately at the beginning of the constructor body if the class has no base class, or immediately following the super call if the class is a derived class.

8.3.3 Automatic Constructors

If a class omits a constructor declaration, an ***automatic constructor*** is provided.

In a class with no `extends` clause, the automatic constructor has no parameters and performs no action other than executing the instance member variable initializers (section 8.4.1), if any.

In a derived class, the automatic constructor has the same parameter list (and possibly overloads) as the base class constructor. The automatically provided constructor first forwards the call to the base class constructor using a call equivalent to

```
BaseClass.apply(this, arguments);
```

and then executes the instance member variable initializers, if any.

8.4 Member Declarations

Member declarations can be member variable declarations, member function declarations, or member accessor declarations.

MemberDeclaration:
MemberVariableDeclaration
MemberFunctionDeclaration
MemberAccessorDeclaration

Member declarations without a `static` modifier are called instance member declarations. Instance member declarations declare properties in the class instance type (section 8.2.4), and must specify names that are unique among all instance member and parameter property declarations in the containing class, with the exception that instance get and set accessor declarations may pairwise specify the same name.

Member declarations with a `static` modifier are called static member declarations. Static member declarations declare properties in the constructor function type (section 8.2.5), and must specify names that are unique among all static member declarations in the containing class, with the exception that static get and set accessor declarations may pairwise specify the same name.

Note that the declaration spaces of instance and static members are separate. Thus, it is possible to have instance and statics members with the same name.

Except for overrides, as described in section 8.2.3, it is an error for a derived class to declare a member with the same name and kind (instance or static) as a base class member.

Every class automatically contains a static member named 'prototype', the type of which is the containing class with type Any substituted for each type parameter. It is an error to explicitly declare a static member with the name 'prototype'.

Below is an example of a class containing both instance and static declarations:

```
class Point {
    constructor(public x: number, public y: number) { }
    public distance(p: Point) {
        var dx = this.x - p.x;
        var dy = this.y - p.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
    static origin = new Point(0, 0);
    static distance(p1: Point, p2: Point) { return p1.distance(p2); }
}
```

The class instance type 'Point' has the members:

```
interface Point {
    x: number;
    y: number;
    distance(p: Point);
}
```

and the constructor function 'Point' has a type corresponding to the declaration:

```
var Point: {
    new(x: number, y: number): Point;
    origin: Point;
    distance(p1: Point, p2: Point): number;
}
```

8.4.1 Member Variable Declarations

A member variable declaration declares an instance member variable or a static member variable.

MemberVariableDeclaration:

PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *TypeAnnotation*$_{opt}$ *Initialiser*$_{opt}$ `;`

The type associated with a member variable declaration is determined in the same manner as an ordinary variable declaration (see section 5.1).

An instance member variable declaration introduces a member in the class instance type and optionally initializes a property on instances of the class. Initializers in instance member variable declarations are executed once for every new instance of the class and are equivalent to assignments to properties of `this` in the constructor. In an initializer expression for an instance member variable, `this` is of the class instance type.

A static member variable declaration introduces a property in the constructor function type and optionally initializes a property on the constructor function object. Initializers in static member variable declarations are executed once when the containing program or module is loaded. In an initializer expression for a static member variable, the type of `this` is the constructor function type excluding construct signatures.

Since instance member variable initializers are equivalent to assignments to properties of `this` in the constructor, the example

```
class Employee {
    public name: string;
    public address: string;
    public retired = false;
    public manager: Employee = null;
    public reports: Employee[] = [];
}
```

is equivalent to

```
class Employee {
    public name: string;
    public address: string;
    public retired: boolean;
    public manager: Employee;
    public reports: Employee[];
    constructor() {
        this.retired = false;
        this.manager = null;
        this.reports = [];
    }
}
```

8.4.2 Member Function Declarations

A member function declaration declares an instance member function or a static member function.

MemberFunctionDeclaration:
    MemberFunctionOverloads$_{opt}$ *MemberFunctionImplementation*

MemberFunctionOverloads:
    MemberFunctionOverload
    MemberFunctionOverloads *MemberFunctionOverload*

MemberFunctionOverload:
    PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `;`

MemberFunctionImplementation:
    PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `{` *FunctionBody* `}`

A member function declaration is processed in the same manner as an ordinary function declaration (section 6.1), except that in a member function `this` has a known type.

All overloads of a member function much have the same accessibility (public or private) and kind (instance or static).

An instance member function declaration declares a property in the class instance type and assigns a function object to a property on the prototype object of the class. In the body of instance member function declaration, `this` is of the class instance type.

A static member function declaration declares a property in the constructor function type and assigns a function object to a property on the constructor function object. In the body of static member function declaration, the type of `this` is the constructor function type excluding construct signatures. (Construct signatures are excluded because a derived class constructor function is not guaranteed to have the same set of construct signatures as a base class constructor function.)

A member function can access overridden base class members using a super property access (section 4.8.2). For example

```
class Point {
    constructor(public x: number, public y: number) { }
    public toString() {
        return "x=" + this.x + " y=" + this.y;
    }
}

class ColoredPoint extends Point {
    constructor(x: number, y: number, public color: string) {
        super(x, y);
    }
    public toString() {
        return super.toString() + " color=" + this.color;
    }
}
```

8.4.3 Member Accessor Declarations

A member accessor declaration declares an instance member accessor or a static member accessor.

MemberAccessorDeclaration:
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *GetAccessor*
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *SetAccessor*

Get and set accessors are processed in the same manner as in an object literal (section 4.5), except that a contextual type is never available in a member accessor declaration.

Accessors for the same member name must specify the same accessibility.

An instance member accessor declaration declares a property in the class instance type and defines a property on the prototype object of the class with a get or set accessor. In the body of an instance member accessor declaration, `this` is of the class instance type.

A static member accessor declaration declares a property in the constructor function type and defines a property on the constructor function object of the class with a get or set accessor. In the body of a static member accessor declaration, the type of `this` is the constructor function type excluding construct signatures.

Get and set accessors are emitted as calls to 'Object.defineProperty' in the generated JavaScript, as described in section 8.5.1.

8.5 Code Generation

This section describes the structure of the JavaScript code generated from TypeScript classes.

8.5.1 Classes Without Extends Clauses

A class with no `extends` clause generates JavaScript equivalent to the following:

```
var <ClassName> = (function () {
    function <ClassName>(<ConstructorParameters>) {
        <DefaultValueAssignments>
        <ParameterPropertyAssignments>
        <MemberVariableAssignments>
        <ConstructorStatements>
    }
    <MemberFunctionStatements>
    <StaticVariableAssignments>
    return <ClassName>;
})();
```

ClassName is the name of the class.

ConstructorParameters is a comma separated list of the constructor's parameter names.

DefaultValueAssignments is a sequence of default property value assignments corresponding to those generated for a regular function declaration, as described in section 6.5.

ParameterPropertyAssignments is a sequence of assignments, one for each parameter property declaration in the constructor, in order they are declared, of the form

```
this.<ParameterName> = <ParameterName>;
```

where *ParameterName* is the name of a parameter property.

MemberVariableAssignments is a sequence of assignments, one for each instance member variable declaration with an initializer, in the order they are declared, of the form

```
this.<MemberName> = <InitializerExpression>;
```

where *MemberName* is the name of the member variable and *InitializerExpression* is the code generated for the initializer expression.

ConstructorStatements is the code generated for the statements specified in the constructor body.

MemberFunctionStatements is a sequence of statements, one for each member function declaration or member accessor declaration, in the order they are declared.

An instance member function declaration generates a statement of the form

```
<ClassName>.prototype.<MemberName> = function (<FunctionParameters>) {
    <DefaultValueAssignments>
    <FunctionStatements>
}
```

and static member function declaration generates a statement of the form

```
<ClassName>.<MemberName> = function (<FunctionParameters>) {
    <DefaultValueAssignments>
    <FunctionStatements>
}
```

where *MemberName* is the name of the member function, and *FunctionParameters*, *DefaultValueAssignments*, and *FunctionStatements* correspond to those generated for a regular function declaration, as described in section 6.5.

A get or set instance member accessor declaration, or a pair of get and set instance member accessor declarations with the same name, generates a statement of the form

```
Object.defineProperty(<ClassName>.prototype, "<MemberName>", {
    get: function () {
        <GetAccessorStatements>
    },
    set: function (<ParameterName>) {
        <SetAccessorStatements>
    },
    enumerable: true,
    configurable: true
};
```

and a get or set static member accessor declaration, or a pair of get and set static member accessor declarations with the same name, generates a statement of the form

```
Object.defineProperty(<ClassName>, "<MemberName>", {
    get: function () {
        <GetAccessorStatements>
    },
    set: function (<ParameterName>) {
        <SetAccessorStatements>
    },
    enumerable: true,
    configurable: true
};
```

where *MemberName* is the name of the member accessor, *GetAccessorStatements* is the code generated for the statements in the get acessor's function body, *ParameterName* is the name of the set accessor parameter, and *SetAccessorStatements* is the code generated for the statements in the set accessor's function body. The 'get' property is included only if a get accessor is declared and the 'set' property is included only if a set accessor is declared.

StaticVariableAssignments is a sequence of statements, one for each static member variable declaration with an initializer, in the order they are declared, of the form

```
<ClassName>.<MemberName> = <InitializerExpression>;
```

where *MemberName* is the name of the static variable, and *InitializerExpression* is the code generated for the initializer expression.

8.5.2 Classes With Extends Clauses

A class with an `extends` clause generates JavaScript equivalent to the following:

```
var <ClassName> = (function (_super) {
    __extends(<ClassName>, _super);
    function <ClassName>(<ConstructorParameters>) {
        <DefaultValueAssignments>
        <SuperCallStatement>
        <ParameterPropertyAssignments>
        <MemberVariableAssignments>
        <ConstructorStatements>
    }
    <MemberFunctionStatements>
    <StaticVariableAssignments>
    return <ClassName>;
})(<BaseClassName>);
```

In addition, the '__extends' function below is emitted at the beginning of the JavaScript source file. It copies all properties from the base constructor function object to the derived constructor function object (in order to inherit static members), and appropriately establishes the 'prototype' property of the derived constructor function object.

```
var __extends = this.__extends || function(d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function f() { this.constructor = d; }
    f.prototype = b.prototype;
    d.prototype = new f();
}
```

BaseClassName is the class name specified in the `extends` clause.

If the class has no explicitly declared constructor, the *SuperCallStatement* takes the form

```
_super.apply(this, arguments);
```

Otherwise the *SuperCallStatement* is present if the constructor function is required to start with a super call, as discussed in section 8.3.2, and takes the form

```
_super.call(this, <SuperCallArguments>)
```

where *SuperCallArguments* is the argument list specified in the super call. Note that this call precedes the code generated for parameter properties and member variables with initializers. Super calls elsewhere in the constructor generate similar code, but the code generated for such calls will be part of the *ConstructorStatements* section.

A super property access in the constructor, an instance member function, or an instance member accessor generates JavaScript equivalent to

```
_super.prototype.<PropertyName>
```

where *PropertyName* is the name of the referenced base class property. When the super property access appears in a function call, the generated JavaScript is equivalent to

```
_super.prototype.<PropertyName>.call(this, <Arguments>)
```

where Arguments is the code generated for the argument list specified in the function call.

A super property access in a static member function or a static member accessor generates JavaScript equivalent to

```
_super.<PropertyName>
```

where *PropertyName* is the name of the referenced base class property. When the super property access appears in a function call, the generated JavaScript is equivalent to

```
_super.<PropertyName>.call(this, <Arguments>)
```

where Arguments is the code generated for the argument list specified in the function call.

9 Enums

An enum type is a distinct subtype of the Number primitive type with an associated set of named constants that define the possible values of the enum type.

9.1 Enum Declarations

An enum declaration declares an ***enum type*** and an ***enum object*** in the containing module.

EnumDeclaration:
`enum` *Identifier* `{` $EnumBody_{opt}$ `}`

The enum type and enum object declared by an *EnumDeclaration* both have the name given by the *Identifier* of the declaration. The enum type is a distinct subtype of the Number primitive type. The enum object is a variable of an anonymous object type containing a set of properties, all of the enum type, corresponding to the values declared for the enum type in the body of the declaration. The enum object's type furthermore includes a numeric index signature with the signature '[x: number]: string'.

The *Identifier* of an enum declaration may not be one of the predefined type names (section 3.6.1).

The example

```
enum Color { Red, Green, Blue }
```

declares a subtype of the Number primitive type called 'Color' and introduces a variable 'Color' with a type that corresponds to the declaration

```
var Color: {
    [x: number]: string;
    Red: Color;
    Green: Color;
    Blue: Color;
};
```

The numeric index signature reflects a "reverse mapping" that is automatically generated in every enum object, as described in section 9.4. The reverse mapping provides a convenient way to obtain the string representation of an enum value. For example

```
var c = Color.Red;
console.log(Color[c]);  // Outputs "Red"
```

9.2 Enum Members

The body of an enum declaration defines zero or more enum members which are the named values of the enum type. Each enum member has an associated numeric value of the primitive type introduced by the enum declaration.

EnumBody:
    ConstantEnumMemberList **,** *opt*
    ConstantEnumMemberList **,** *ComputedEnumMemberList* **,** *opt*
    ComputedEnumMemberList **,** *opt*

ConstantEnumMemberList:
    ConstantEnumMember
    ConstantEnumMemberList **,** *ConstantEnumMember*

ConstantEnumMember:
    PropertyName
    PropertyName **=** *IntegerLiteral*

IntegerLiteral:
    SignedInteger
    HexIntegerLiteral

ComputedEnumMemberList:
    ComputedEnumMember
    ComputedEnumMemberList **,** *ComputedEnumMember*

ComputedEnumMember:
    PropertyName **=** *AssignmentExpression*

Enum members are either ***constant members*** or ***computed members***. Constant members have known constant values that may be substituted in place of references to the members at compile-time. Computed members have values that are computed at run-time and not known at compile-time.

TODO: Decide whether references to constant members are always replaced with their values or whether this is a compiler option.

The body of an enum declaration consists of a list of zero or more constant members followed by zero or more computed members.

A constant member declared with an integer literal value is assigned that value. A constant member declared without an integer literal value is assigned the value of the preceding constant member plus one, or the value zero if it is the first member in the enum body. The values of constant members are computed at compile-time and may be substituted for references to the members in the generated JavaScript code.

Expressions specified for computed members must produce values of type Any, the Number primitive type, or an enum type. The values of computed members are not known at compile-time and no substitution is performed for references to computed members.

In the example

```
enum Test {
    A,
    B,
    C = 10,
    D,
    E = Math.floor(Math.random() * 1000)
}
```

'A', 'B', 'C', and 'D' are constant members with values 0, 1, 10, and 11 respectively, and 'E' is a computed member.

In the example

```
enum Style {
    None = 0,
    Bold = 1,
    Italic = 2,
    Underline = 4,
    Emphasis = Style.Bold | Style.Italic,
    Hyperlink = Style.Bold | Style.Underline
}
```

the first four members are constant members and the last two are computed members. Note that computed member declarations can reference previously declared members. Also, because enums are subtypes of the Number primitive type, numeric operators, such as the bitwise OR operator, can be used to compute enum values.

9.3 Declaration Merging

Enums are "open-ended" and enum declarations with the same qualified name relative to a common root (as defined in section 2.3) define a single enum type and contribute to a single enum object.

It isn't possible for one enum declaration to continue the automatic numbering sequence of another, and when an enum type has multiple declarations, only one declaration is permitted to omit a value for the first member.

9.4 Code Generation

An enum declaration generates JavaScript equivalent to the following:

```
var <EnumName>;
(function (<EnumName>) {
    <EnumMemberAssignments>
})(<EnumName>||(<EnumName>={}));
```

EnumName is the name of the enum.

EnumMemberAssignments is a sequence of assignments, one for each enum member, in order they are declared, of the form

```
<EnumName>[<EnumName>.<MemberName> = <MemberValue>] = "<MemberName>";
```

where *MemberName* is the name of the enum member and *MemberValue* is the assigned constant value or the code generated for the computed value expression.

TODO: In general, when generating code for a property access, use quoted names and bracket notation for non-identifier member names.

For example, the 'Color' enum example from section 9.1 generates the following JavaScript:

```
var Color;
(function (Color) {
    Color[Color.Red = 0] = "Red";
    Color[Color.Green = 1] = "Green";
    Color[Color.Blue = 2] = "Blue";
})(Color||(Color={}));
```

10 Internal Modules

An internal module is a named container of statements and declarations. An internal module represents both a namespace and a singleton module instance. The namespace contains named types and other namespaces, and the singleton module instance contains properties for the module's exported members. The body of an internal module corresponds to a function that is executed once, thereby providing a mechanism for maintaining local state with assured isolation.

10.1 Module Declarations

An internal module declaration declares a namespace name and, in the case of an instantiated module, a member name in the containing module.

ModuleDeclaration:
`module` *IdentifierPath* `{` *ModuleBody* `}`

IdentifierPath:
Identifier
IdentifierPath `.` *Identifier*

Internal modules are either ***instantiated*** or ***non-instantiated***. A non-instantiated module is an internal module containing only interface types and other non-instantiated modules. An instantiated module is an internal module that doesn't meet this definition. In intuitive terms, an instantiated module is one for which a module object instance is created, whereas a non-instantiated module is one for which no code is generated.

When a module identifier is referenced as a *ModuleName* (section 3.6.2) it denotes a container of module and type names, and when a module identifier is referenced as a *PrimaryExpression* (section 4.3) it denotes the singleton module instance. For example:

```
module M {
    export interface P { x: number; y: number; }
    export var a = 1;
}

var p: M.P;             // M used as ModuleName
var m = M;              // M used as PrimaryExpression
var x1 = M.a;           // M used as PrimaryExpression
var x2 = m.a;           // Same as M.a
var q: m.P;             // Error
```

Above, when 'M' is used as a *PrimaryExpression* it denotes an object instance with a single member 'a' and when 'M' is used as a *ModuleName* it denotes a container with a single type member 'P'. The final line in the example is an error because 'm' is a variable which cannot be referenced in a type name.

If the declaration of 'M' above had excluded the exported variable 'a', 'M' would be a non-instantiated module and it would be an error to reference 'M' as a *PrimaryExpression*.

An internal module declaration that specifies an *IdentifierPath* with more than one identifier is equivalent to a series of nested single-identifier internal module declarations where all but the outermost are automatically exported. For example:

```
module A.B.C {
    export var x = 1;
}
```

corresponds to

```
module A {
    export module B {
        export module C {
            export var x = 1;
        }
    }
}
```

10.2 Module Body

The body of an internal module corresponds to a function that is executed once to initialize the module instance.

ModuleBody:
    ModuleElements$_{opt}$

ModuleElements:
    ModuleElement
    ModuleElements *ModuleElement*

ModuleElement:
    Statement
    `export`$_{opt}$ *VariableDeclaration*
    `export`$_{opt}$ *FunctionDeclaration*
    `export`$_{opt}$ *ClassDeclaration*
    `export`$_{opt}$ *InterfaceDeclaration*
    `export`$_{opt}$ *EnumDeclaration*
    `export`$_{opt}$ *ModuleDeclaration*
    `export`$_{opt}$ *ImportDeclaration*

Each module body has a declaration space for local variables (including functions, modules, class constructor functions, and enum objects), a declaration space for local named types (classes, interfaces,

and enums), and a declaration space for local namespaces (containers of named types). Every declaration (whether local or exported) in a module contributes to one or more of these declaration spaces.

10.3 Import Declarations

Import declarations are used to create local aliases for entities in other modules.

ImportDeclaration:
`import` *Identifier* `=` *EntityName* `;`

EntityName:
ModuleName
ModuleName `.` *Identifier*

An *EntityName* consisting of a single identifier is resolved as a *ModuleName* and is thus required to reference an internal module. The resulting local alias references the given internal module and is itself classified as an internal module.

An *EntityName* consisting of more than one identifier is resolved as a *ModuleName* followed by an identifier that names one or more exported entities in the given module. The resulting local alias has all the meanings and classifications of the referenced entity or entities. (As many as three distinct meanings are possible for an entity name—namespace, type, and member.) In effect, it is as if the imported entity or entities were declared locally with the local alias name.

In the example

```
module A {
    export interface X { s: string }
    export var X: X;
}

module B {
    interface A { n: number }
    import Y = A;    // Alias only for module A
    import Z = A.X;  // Alias for both type and member A.X
    var v: Z = Z;
}
```

within 'B', 'Y' is an alias only for module 'A' and not the local interface 'A', whereas 'Z' is an alias for all exported meanings of 'A.X', thus denoting both an interface type and a variable.

If the *ModuleName* portion of an *EntityName* references an instantiated module, the *ModuleName* is required to reference the module instance when evaluated as an expression. In the example

```
module A {
    export interface X { s: string }
}
```

```
module B {
    var A = 1;
    import Y = A;
}
```

'Y' is a local alias for the non-instantiated module 'A'. If the declaration of 'A' is changed such that 'A' becomes an instantiated module, for example by including a variable declaration in 'A', the import statement in 'B' above would be an error because the expression 'A' doesn't reference the module instance of module 'A'.

When an import statement includes an export modifier, all meanings of the local alias are exported.

TODO: Specify the exact restrictions on import declarations referencing other import declarations. We minimally want to disallow circular references.

10.4 Export Declarations

An export declaration declares an externally accessible module member. An export declaration is simply a regular declaration prefixed with the keyword `export`.

Exported class, interface, and enum types can be accessed as a *TypeName* (section 3.6.2) of the form *M.T*, where *M* is a reference to the containing module and *T* is the exported type name. Likewise, as part of a *TypeName*, exported modules can be accessed as a *ModuleName* of the form *M.N*, where *M* is a reference to the containing module and *N* is the exported module.

Exported variable, function, class, enum, module, and import alias declarations become properties on the module instance and together establish the module's ***instance type***. This unnamed type has the following members:

- A property for each exported variable declaration.
- A property of a function type for each exported function declaration.
- A property of a constructor type for each exported class declaration.
- A property of an object type for each exported enum declaration.
- A property of an object type for each exported instantiated module declaration.
- A property for each exported import alias that references a variable, function, class, enum, or instantiated module.

An exported member depends on a (possibly empty) set of named types (section 3.5). Those named types must be at least as accessible as the exported member, or otherwise an error occurs.

The named types upon which a member depends are the named types occurring in the transitive closure of the ***directly depends on*** relationship defined as follows:

- A variable directly depends on the *Type* specified in its type annotation.
- A function directly depends on each *Type* specified in a parameter or return type annotation.

- A class directly depends on each *TypeReference* specified as a base class or implemented interface, and each *Type* specified in a constructor parameter type annotation, member variable type annotation, member function parameter or return type annotation, member accessor parameter or return type annotation, or index signature type annotation.
- An interface directly depends on each *TypeReference* specified as a base interface and on the *ObjectType* specified as its body.
- A module directly depends on its exported members.
- A *Type* or *ObjectType* directly depends on every *TypeReference* that occurs within the type at any level of nesting.
- A *TypeReference* directly depends on the type it references and on each *Type* specified as a type argument.

A named type *T* having a root module *R* (section 2.3) is said to be ***at least as accessible as*** a member *M* if

- *R* is the global module or an external module, or
- *R* is an internal module in the parent module chain of *M*.

In the example

```
interface A { x: string; }

module M {
    export interface B { x: A; }
    export interface C { x: B; }
    export function foo(c: C) { … }
}
```

the 'foo' function depends upon the named types 'A', 'B', and 'C'. In order to export 'foo' it is necessary to also export 'B' and 'C' as they otherwise would not be at least as accessible as 'foo'. The 'A' interface is already at least as accessible as 'foo' because it is declared in a parent module of foo's module.

10.5 Declaration Merging

Internal modules are "open-ended" and internal module declarations with the same qualified name relative to a common root (as defined in section 2.3) contribute to a single module. For example, the following two declarations of a module outer might be located in separate source files.

File a.ts:

```
module outer {
    var local = 1;           // Non-exported local variable
    export var a = local;    // outer.a
    export module inner {
        export var x = 10;   // outer.inner.x
    }
}
```

File b.ts:

```
module outer {
    var local = 2;           // Non-exported local variable
    export var b = local;    // outer.b
    export module inner {
        export var y = 20;   // outer.inner.y
    }
}
```

Assuming the two source files are part of the same program, the two declarations will have the global module as their common root and will therefore contribute to the same module instance, the instance type of which will be:

```
{
    a: number;
    b: number;
    inner: {
        x: number;
        y: number;
    };
}
```

Declaration merging does not apply to local aliases created by import declarations. In other words, it is not possible have an import declaration and a module declaration for the same name within the same module body.

Declaration merging also extends to internal module declarations with the same qualified name relative to a common root as a function, class, or enum declaration:

- When merging a function and an internal module, the type of the function object is merged with the instance type of the module. In effect, the overloads or implementation of the function provide the call signatures and the exported members of the module provide the properties of the combined type.
- When merging a class and an internal module, the type of the constructor function object is merged with the instance type of the module. In effect, the overloads or implementation of the class constructor provide the construct signatures, and the static members of the class and exported members of the module provide the properties of the combined type. It is an error to have static class members and exported module members with the same name.
- When merging an enum and an internal module, the type of the enum object is merged with the instance type of the module. In effect, the members of the enum and the exported members of the module provide the properties of the combined type. It is an error to have enum members and exported module members with the same name.

When merging a non-ambient function or class declaration and a non-ambient internal module declaration, the function or class declaration must be located prior to the internal module declaration in

the same source file. This ensures that the shared object instance is created as a function object. (While it is possible to add properties to an object after its creation, it is not possible to make an object "callable" after the fact.)

The example

```
interface Point {
    x: number;
    y: number;
}

function point(x: number, y: number): Point {
    return { x: x, y: y };
}

module point {
    export var origin = point(0, 0);
    export function equals(p1: Point, p2: Point) {
        return p1.x == p2.x && p1.y == p2.y;
    }
}

var p1 = point(0, 0);
var p2 = point.origin;
var b = point.equals(p1, p2);
```

declares 'point' as a function object with two properties, 'origin' and 'equals'. Note that the module declaration for 'point' is located after the function declaration.

10.6 Code Generation

An internal module generates JavaScript code that is equivalent to the following:

```
var <ModuleName>;
(function(<ModuleName>) {
    <ModuleStatements>
})(<ModuleName>||(<ModuleName>={}));
```

where *ModuleName* is the name of the module and *ModuleStatements* is the code generated for the statements in the module body. The *ModuleName* function parameter may be prefixed with one or more underscore characters to ensure the name is unique within the function body. Note that the entire module is emitted as an anonymous function that is immediately executed. This ensures that local variables are in their own lexical environment isolated from the surrounding context. Also note that the generated function doesn't create and return a module instance, but rather it extends the existing instance (which may have just been created in the function call). This ensures that internal modules can extend each other.

An import statement generates code of the form

```
var <Alias> = <EntityName>;
```

This code is emitted only if the imported entity is referenced as a *PrimaryExpression* somewhere in the body of the importing module. If an imported entity is referenced only as a *TypeName* or *ModuleName*, nothing is emitted. This ensures that types declared in one internal module can be referenced through an import alias in another internal module with no run-time overhead.

When a variable is exported, all references to the variable in the body of the module are replaced with

```
<ModuleName>.<VariableName>
```

This effectively promotes the variable to be a property on the module instance and ensures that all references to the variable become references to the property.

When a function, class, enum, or module is exported, the code generated for the entity is followed by an assignment statement of the form

```
<ModuleName>.<EntityName> = <EntityName>;
```

This copies a reference to the entity into a property on the module instance.

11 Source Files and External Modules

TypeScript implements external modules that are closely aligned with those proposed for ECMAScript 6 and supports code generation targeting CommonJS and AMD module systems.

NOTE: TypeScript currently doesn't support the full proposed capabilities of the ECMAScript 6 import and export syntax. We expect to align more closely on the syntax as the ECMAScript 6 specification evolves.

11.1 Source Files

A TypeScript ***program*** consists of one or more source files that are either ***implementation source files*** or ***declaration source files***. Source files with extension '.ts' are *ImplementationSourceFiles* containing statements and declarations. Source files with extension '.d.ts' are *DeclarationSourceFiles* containing declarations only. Declaration source files are a strict subset of implementation source files.

SourceFile:
    ImplementationSourceFile
    DeclarationSourceFile

ImplementationSourceFile:
    ImplementationElements$_{opt}$

ImplementationElements:
    ImplementationElement
    ImplementationElements *ImplementationElement*

ImplementationElement:
    ModuleElement
    ExportAssignment
    `export`$_{opt}$ *ExternalImportDeclaration*
    `export`$_{opt}$ *AmbientDeclaration*

DeclarationSourceFile:
    DeclarationElements$_{opt}$

DeclarationElements:
    DeclarationElement
    DeclarationElements *DeclarationElement*

DeclarationElement:

`export`$_{opt}$ *InterfaceDeclaration*

`export`$_{opt}$ *ImportDeclaration*

ExportAssignment

`export`$_{opt}$ *ExternalImportDeclaration*

`export`$_{opt}$ *AmbientDeclaration*

When a TypeScript program is compiled, all of the program's source files are processed together. Statements and declarations in different source files can depend on each other, possibly in a circular fashion. By default, a JavaScript output file is generated for each implementation source file in a compilation, but no output is generated from declaration source files.

The source elements permitted in a TypeScript implementation source file are a superset of those supported by JavaScript. Specifically, TypeScript extends the JavaScript grammar's existing *VariableDeclaration* (section 5.1) and *FunctionDeclaration* (section 6.1) productions, and adds *InterfaceDeclaration* (section 7.1), *ClassDeclaration* (section 8.1), *EnumDeclaration* (section 9.1), *ModuleDeclaration* (section 10.1), *ImportDeclaration* (section 10.3), *ExternalImportDeclaration* (section 11.2.2), *ExportAssignment* (section 11.2.4), and *AmbientDeclaration* (section 12.1) productions.

Declaration source files are restricted to contain declarations only. Declaration source files can be used to declare the static type information associated with existing JavaScript code in an adjunct manner. They are entirely optional but enable the TypeScript compiler and tools to provide better verification and assistance when integrating existing JavaScript code and libraries in a TypeScript application.

Implementation and declaration source files that contain no import or export declarations form the single ***global module***. Entities declared in the global module are in scope everywhere in a program. Initialization order of the source files that make up the global module ultimately depends on the order in which the generated JavaScript files are loaded at run-time (which, for example, may be controlled by <script/> tags that reference the generated JavaScript files).

Implementation and declaration source files that contain at least one import or export declaration are considered separate ***external modules***. Entities declared in an external module are in scope only in that module, but exported entities can be imported into other modules using import declarations. Initialization order of external modules is determined by the module loader being and is not specified by the TypeScript language. However, it is generally the case that non-circularly dependent modules are automatically loaded and initialized in the correct order.

External modules can additionally be declared using *AmbientModuleDeclarations* in the global module that directly specify the external module names as string literals. This is described further in section 12.1.6.

11.1.1 Source Files Dependencies

The TypeScript compiler automatically determines a source file's dependencies and includes those dependencies in the program being compiled. The determination is made from "reference comments" and external import declarations as follows:

- A comment of the form /// <reference path="..."/> adds a dependency on the source file specified in the path argument. The path is resolved relative to the directory of the containing source file.
- An external import declaration that specifies a relative external module name (section 11.2.1) resolves the name relative to the directory of the containing source file. If a source file with the resulting path and file extension '.ts' exists, that file is added as a dependency. Otherwise, if a source file with the resulting path and file extension '.d.ts' exists, that file is added as a dependency.
- An external import declaration that specifies a top-level external module name (section 11.2.1) resolves the name in a host dependent manner (typically by resolving the name relative to a module name space root or searching for the name in a series of directories). If a source file with extension '.ts' or '.d.ts' corresponding to the reference is located, that file is added as a dependency.

Any files included as dependencies in turn have their references analyzed in a transitive manner until all dependencies have been determined.

11.2 External Modules

External modules are separately loaded bodies of code referenced using external module names. External modules can be likened to functions that are loaded and executed once to initialize their associated module instance. Entities declared in an external module are private and inaccessible elsewhere unless they are exported.

External modules are written as separate source files that contain at least one import or export declaration. Specifically, if a source file contains at least one *ExternalImportDeclaration* or *ExportAssignment*, or at least one top-level *VariableDeclaration*, *FunctionDeclaration*, *ClassDeclaration*, *InterfaceDeclaration*, *EnumDeclaration*, *ModuleDeclaration*, *ImportDeclaration*, or *AmbientDeclaration* that specifies an `export` modifier, that source file is considered an external module; otherwise, the source file is considered part of the global module.

Below is an example of two external modules written in separate source files.

File main.ts:

```
import log = require("log");
log.message("hello");
```

File log.ts:

```
export function message(s: string) {
    console.log(s);
}
```

The import declaration in the 'main' module references the 'log' module and compiling the 'main.ts' file causes the 'log.ts' file to also be compiled as part of the program. At run-time, the import declaration

loads the 'log' module and produces a reference to its module instance through which it is possible to reference the exported function.

TypeScript supports two patterns of JavaScript code generation for external modules: The CommonJS Modules pattern (section 11.2.5), typically used by server frameworks such as node.js, and the Asynchronous Module Definition (AMD) pattern (section 11.2.6), an extension to CommonJS Modules that permits asynchronous module loading, as is typical in browsers. The desired module code generation pattern is selected through a compiler option and does not affect the TypeScript source code. Indeed, it is possible to author external modules that can be compiled for use both on the server side (e.g. using node.js) and on the client side (using an AMD compliant loader) with no changes to the TypeScript source code.

11.2.1 External Module Names

External modules are identified and referenced using external module names. The following definition is copied from the CommonJS Modules 1.0 specification, with the term 'module identifier' changed to 'external module name':

- An external module name is a string of "terms" delimited by forward slashes.
- A term must be a camelCase identifier, ".", or "..".
- External module names may not have file-name extensions like ".js".
- External module names may be "relative" or "top-level". An external module name is "relative" if the first term is "." or "..".
- Top-level names are resolved off the conceptual module name space root.
- Relative names are resolved relative to the name of the module in which they occur.

For purposes of resolving external module references, TypeScript associates a file path with every external module. The file path is simply the path of the module's source file without the file extension. For example, an external module contained in the source file 'C:\src\lib\io.ts' has the file path 'C:/src/lib/io' and an external module contained in the source file 'C:\src\ui\editor.d.ts' has the file path 'C:/src/ui/editor'.

An external module name in an import declaration is resolved as follows:

- If the import declaration specifies a relative external module name, the name is resolved relative to the directory of the referencing module's file path. The program must contain a module with the resulting file path or otherwise an error occurs. For example, in a module with the file path 'C:/src/ui/main', the external module names './editor' and '../lib/io' reference modules with the file paths 'C:/src/ui/editor' and 'C:/src/lib/io'.
- If the import declaration specifies a top-level external module name and the program contains an *AmbientExternalModuleDeclaration* (section 12.1.6) with a string literal that specifies that exact name, then the import declaration references that ambient external module.
- If the import declaration specifies a top-level external module name and the program contains no *AmbientExternalModuleDeclaration* (section 12.1.6) with a string literal that specifies that exact name, the name is resolved in a host dependent manner (for example by considering the name relative to a module name space root). If a matching module cannot be found an error occurs.

11.2.2 External Import Declarations

External import declarations are used to import external modules and create local aliases by which they may be referenced.

ExternalImportDeclaration:
`import` *Identifier* `=` *ExternalModuleReference* `;`

ExternalModuleReference:
`require` `(` *StringLiteral* `)`

The string literal specified in an *ExternalModuleReference* is interpreted as an external module name (section 11.2.1).

An external import declaration introduces a local identifier that references a given external module. The local identifier becomes an alias for, and is classified exactly like, the entity or entities exported from the referenced external module. Specifically, if the referenced external module contains no export assignment the identifier is classified as a module, and if the referenced external module contains an export assignment the identifier is classified exactly like the entity or entities named in the export assignment.

11.2.3 Export Declarations

An external module that contains no export assignment (section 11.2.4) exports an entity classified as a module. Similarly to an internal module, export declarations (section 10.4) in the external module are used to declare the members of this entity.

Unlike a non-instantiated internal module (section 10.1), an external module containing only interface types and non-instantiated internal modules still has a module instance associated with it, albeit one with no members.

If an external module contains an export assignment it is an error for the external module to also contain export declarations. The two types of exports are mutually exclusive.

11.2.4 Export Assignments

An export assignment designates a module member as the entity to be exported in place of the external module itself.

ExportAssignment:
`export` `=` *Identifier* `;`

The *Identifier* of an export assignment must name one or more entities declared at the top level in the external module. When an external module containing an export assignment is imported, the local alias introduced by the external import declaration takes on all meanings of the identifier named in the export assignment.

It is an error for an external module to contain more than one export assignment.

Assume the following example resides in the file 'point.ts':

```
export = Point;

class Point {
    constructor(public x: number, public y: number) { }
    static origin = new Point(0, 0);
}
```

When 'point.ts' is imported in another external module, the import alias references the exported class and can be used both as a type and as a constructor function:

```
import Pt = require("point.ts");

var p1 = new Pt(10, 20);
var p2 = Pt.origin;
```

Note that there is no requirement that the import alias use the same name as the exported entity.

11.2.5 CommonJS Modules

The CommonJS Modules definition specifies a methodology for writing JavaScript modules with implied privacy, the ability to import other modules, and the ability to explicitly export members. A CommonJS compliant system provides a 'require' function that can be used to synchronously load other external modules to obtain their singleton module instance, as well as an 'exports' variable to which a module can add properties to define its external API.

The 'main' and 'log' example from section 11.2 above generates the following JavaScript code when compiled for the CommonJS Modules pattern:

File main.js:

```
var log = require("log");
log.message("hello");
```

File log.js:

```
exports.message = function(s) {
    console.log(s);
}
```

An external import declaration is represented in the generated JavaScript as a variable initialized by a call to the 'require' function provided by the module system host. A variable declaration and 'require' call is emitted for a particular imported module only if the imported module is referenced as a *PrimaryExpression* somewhere in the body of the importing module. If an imported module is referenced only as a *ModuleName*, nothing is emitted.

An example:

File geometry.ts:

```
export interface Point { x: number; y: number };

export function point(x: number, y: number): Point {
    return { x: x, y: y };
}
```

File game.ts:

```
import g = require("geometry");
var p = g.point(10, 20);
```

The 'game' module references the imported 'geometry' module in an expression (through its alias 'g') and a 'require' call is therefore included in the emitted JavaScript:

```
var g = require("geometry");
var p = g.point(10, 20);
```

Had the 'game' module instead been written to only reference 'geometry' in a type position

```
import g = require("geometry");
var p: g.Point = { x: 10, y: 20 };
```

the emitted JavaScript would have no dependency on the 'geometry' module and would simply be

```
var p = { x: 10, y: 20 };
```

11.2.6 AMD Modules

The Asynchronous Module Definition (AMD) specification extends the CommonJS Modules specification with a pattern for authoring asynchronously loadable modules with associated dependencies. Using the AMD pattern, modules are emitted as calls to a global 'define' function taking an array of dependencies, specified as external module names, and a callback function containing the module body. The global 'define' function is provided by including an AMD compliant loader in the application. The loader arranges to asynchronously load the module's dependencies and, upon completion, calls the callback function passing resolved module instances as arguments in the order they were listed in the dependency array.

The "main" and "log" example from above generates the following JavaScript code when compiled for the AMD pattern.

File main.js:

```
define(["require", "exports", "log"], function(require, exports, log) {
    log.message("hello");
}
```

File log.js:

```
define(["require", "exports"], function(require, exports) {
    exports.message = function(s) {
        console.log(s);
    }
}
```

The special ‘require’ and ‘exports’ dependencies are always present. Additional entries are added to the dependencies array and the parameter list as required to represent imported external modules. Similar to the code generation for CommonJS Modules, a dependency entry is generated for a particular imported module only if the imported module is referenced as a *PrimaryExpression* somewhere in the body of the importing module. If an imported module is referenced only as a *ModuleName*, no dependency is generated for that module.

11.3 Code Generation

TODO: Finish this section.

12 Ambients

Ambient declarations are used to provide static typing over existing JavaScript code. Ambient declarations differ from regular declarations in that no JavaScript code is emitted for them. Instead of introducing new variables, functions, classes, enums, or modules, ambient declarations provide type information for entities that exist "ambiently" and are included in a program by external means, for example by referencing a JavaScript library in a <script/> tag.

TODO: Finish this chapter.

12.1 Ambient Declarations

Ambient declarations are written using the `declare` keyword and can declare variables, functions, classes, enums, internal modules, or external modules.

AmbientDeclaration:
`declare` *AmbientVariableDeclaration*
`declare` *AmbientFunctionDeclaration*
`declare` *AmbientClassDeclaration*
`declare` *AmbientEnumDeclaration*
`declare` *AmbientModuleDeclaration*
`declare` *AmbientExternalModuleDeclaration*

Ambient declarations are only permitted at the top-level in a source file (section 11.1).

12.1.1 Ambient Variable Declarations

An ambient variable declaration introduces a variable in the containing declaration space.

AmbientVariableDeclaration:
`var` *Identifier* *TypeAnnotation*$_{opt}$ `;`

An ambient variable declaration may optionally include a type annotation. If no type annotation is present, the variable is assumed to have type Any.

An ambient variable declaration does not permit an initializer expression to be present.

12.1.2 Ambient Function Declarations

An ambient function declaration introduces a function in the containing declaration space.

AmbientFunctionDeclaration:
`function` *Identifier* *CallSignature* `;`

Ambient functions may be overloaded by specifying multiple ambient function declarations with the same name, but it is an error to declare multiple overloads that are considered identical (section 3.8.1) or differ only in their return types.

Ambient function declarations cannot specify a function bodies and do not permit default parameter values.

12.1.3 Ambient Class Declarations

An ambient class declaration declares a class instance type and a constructor function in the containing module.

AmbientClassDeclaration:
`class` *Identifier* *TypeParameters*$_{opt}$ *ClassHeritage* `{` *AmbientClassBody* `}`

AmbientClassBody:
AmbientClassBodyElements$_{opt}$

AmbientClassBodyElements:
AmbientClassBodyElement
AmbientClassBodyElements *AmbientClassBodyElement*

AmbientClassBodyElement:
AmbientConstructorDeclaration
AmbientMemberDeclaration
IndexSignature

AmbientConstructorDeclaration:
`constructor` `(` *ParameterList*$_{opt}$ `)` `;`

AmbientMemberDeclaration:
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *TypeAnnotation*$_{opt}$ `;`
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `;`

12.1.4 Ambient Enum Declarations

An ambient enum declaration declares an enum type and an enum object in the containing module.

AmbientEnumDeclaration:
`enum` *Identifier* `{` *AmbientEnumBody*$_{opt}$ `}`

AmbientEnumBody:
AmbientEnumMemberList `,`$_{opt}$

AmbientEnumMemberList:
AmbientEnumMember
AmbientEnumMemberList `,` *AmbientEnumMember*

AmbientEnumMember:
PropertyName
PropertyName = *IntegerLiteral*

An *AmbientEnumMember* that includes an *IntegerLiteral* value is considered a constant member. An *AmbientEnumMember* with no *IntegerLiteral* value is considered a computed member.

12.1.5 Ambient Module Declarations

An ambient module declaration declares an internal module.

AmbientModuleDeclaration:
`module` *IdentifierPath* `{` *AmbientModuleBody* `}`

AmbientModuleBody:
AmbientModuleElements$_{opt}$

AmbientModuleElements:
AmbientModuleElement
AmbientModuleElements *AmbientModuleElement*

AmbientModuleElement:
`export`$_{opt}$ *AmbientVariableDeclaration*
`export`$_{opt}$ *AmbientFunctionDeclaration*
`export`$_{opt}$ *AmbientClassDeclaration*
`export`$_{opt}$ *InterfaceDeclaration*
`export`$_{opt}$ *AmbientEnumDeclaration*
`export`$_{opt}$ *AmbientModuleDeclaration*
`export`$_{opt}$ *ImportDeclaration*

An *AmbientModuleElement* always declares an exported entity regardless of whether the optional `export` modifier is included.

12.1.6 Ambient External Module Declarations

An *AmbientExternalModuleDeclaration* declares an external module. This type of declaration is permitted only in the global module. The *StringLiteral* must specify a top-level external module name. Relative external module names are not permitted.

AmbientExternalModuleDeclaration:
`module` *StringLiteral* `{` *AmbientExternalModuleBody* `}`

AmbientExternalModuleBody:
AmbientExternalModuleElements$_{opt}$

AmbientExternalModuleElements:
AmbientExternalModuleElement
AmbientExternalModuleElements *AmbientExternalModuleElement*

AmbientExternalModuleElement:
AmbientModuleElement
ExportAssignment
`export`$_{opt}$ *ExternalImportDeclaration*

An *ExternalImportDeclaration* in an *AmbientExternalModuleDeclaration* may reference other external modules only through top-level external module names. Relative external module names are not permitted.

If an ambient external module declaration includes an export assignment, it is an error for any of the declarations with the module to specify an `export` modifier. If an ambient external module declaration contains no export assignment, entities declared in the module are exported regardless of whether their declarations include the optional `export` modifier.

A Grammar

This appendix contains a summary of the grammar found in the main document. As described in section 2.1, the TypeScript grammar is a superset of the grammar defined in the ECMAScript Language Specification (specifically, the ECMA-262 Standard, 5th Edition) and this appendix lists only productions that are new or modified from the ECMAScript grammar.

A.1 Types

TypeParameters:
    < *TypeParameterList* >

TypeParameterList:
    TypeParameter
    TypeParameterList , *TypeParameter*

TypeParameter:
    Identifier *Constraint*$_{opt}$

Constraint:
    `extends` *TypeReference*

Type:
    PredefinedType
    TypeReference
    TypeQuery
    TypeLiteral

PredefinedType:
    `any`
    `number`
    `boolean`
    `string`
    `void`

TypeReference:
    TypeName *TypeArguments*$_{opt}$

TypeName:
    Identifier
    ModuleName . *Identifier*

ModuleName:
    Identifier
    ModuleName . *Identifier*

TypeArguments:
    `<` *TypeArgumentList* `>`

TypeArgumentList:
    TypeArgument
    TypeArgumentList `,` *TypeArgument*

TypeArgument:
    Type

TypeQuery:
    `typeof` *TypeQueryExpression*

TypeQueryExpression:
    Identifier
    TypeQueryExpression `.` *IdentifierName*

TypeLiteral:
    ObjectType
    ArrayType
    FunctionType
    ConstructorType

ArrayType:
    PredefinedType `[ ]`
    TypeReference `[ ]`
    ObjectType `[ ]`
    ArrayType `[ ]`

FunctionType:
    $TypeParameters_{opt}$ `(` $ParameterList_{opt}$ `) =>` *Type*

ConstructorType:
    `new` $TypeParameters_{opt}$ `(` $ParameterList_{opt}$ `) =>` *Type*

ObjectType:
    `{` $TypeBody_{opt}$ `}`

TypeBody:
    TypeMemberList $;_{opt}$

TypeMemberList:
    TypeMember
    TypeMemberList `;` *TypeMember*

TypeMember:

PropertySignature
CallSignature
ConstructSignature
IndexSignature
MethodSignature

PropertySignature:

PropertyName `?`$_{opt}$ *TypeAnnotation*$_{opt}$

PropertyName:

IdentifierName
StringLiteral
NumericLiteral

CallSignature:

TypeParameters$_{opt}$ `(` *ParameterList*$_{opt}$ `)` *TypeAnnotation*$_{opt}$

ParameterList:

RequiredParameterList
OptionalParameterList
RestParameter
RequiredParameterList `,` *OptionalParameterList*
RequiredParameterList `,` *RestParameter*
OptionalParameterList `,` *RestParameter*
RequiredParameterList `,` *OptionalParameterList* `,` *RestParameter*

RequiredParameterList:

RequiredParameter
RequiredParameterList `,` *RequiredParameter*

RequiredParameter:

PublicOrPrivate$_{opt}$ *Identifier* *TypeAnnotation*$_{opt}$
Identifier `:` *StringLiteral*

PublicOrPrivate:

`public`
`private`

OptionalParameterList:

OptionalParameter
OptionalParameterList `,` *OptionalParameter*

OptionalParameter:

PublicOrPrivate$_{opt}$ *Identifier* `?` *TypeAnnotation*$_{opt}$

PublicOrPrivate$_{opt}$ *Identifier* *TypeAnnotation*$_{opt}$ *Initialiser*

RestParameter:

`...` *Identifier* *TypeAnnotation*$_{opt}$

ConstructSignature:

`new` *TypeParameters*$_{opt}$ `(` *ParameterList*$_{opt}$ `)` *TypeAnnotation*$_{opt}$

IndexSignature:

`[` *Identifier* `:` `string` `]` *TypeAnnotation*

`[` *Identifier* `:` `number` `]` *TypeAnnotation*

MethodSignature:

PropertyName `?`$_{opt}$ *CallSignature*

A.2 Expressions

PropertyAssignment: (Modified)

PropertyName `:` *AssignmentExpression*

PropertyName *CallSignature* `{` *FunctionBody* `}`

GetAccessor

SetAccessor

GetAccessor:

`get` *PropertyName* `(` `)` *TypeAnnotation*$_{opt}$ `{` *FunctionBody* `}`

SetAccessor:

`set` *PropertyName* `(` *Identifier* *TypeAnnotation*$_{opt}$ `)` `{` *FunctionBody* `}`

CallExpression: (Modified)

...

`super` `(` *ArgumentList*$_{opt}$ `)`

`super` `.` *IdentifierName*

FunctionExpression: (Modified)

`function` *Identifier*$_{opt}$ *CallSignature* `{` *FunctionBody* `}`

AssignmentExpression: (Modified)

...

ArrowFunctionExpression

ArrowFunctionExpression:

ArrowFormalParameters `=>` *Block*

ArrowFormalParameters `=>` *AssignmentExpression*

ArrowFormalParameters:
CallSignature
Identifier

Arguments: (Modified)
TypeArguments$_{opt}$ (*ArgumentList*$_{opt}$)

UnaryExpression: (Modified)
...
< *Type* > *UnaryExpression*

A.3 Statements

VariableDeclaration: (Modified)
Identifier *TypeAnnotation*$_{opt}$ *Initialiser*$_{opt}$

VariableDeclarationNoIn: (Modified)
Identifier *TypeAnnotation*$_{opt}$ *InitialiserNoIn*$_{opt}$

TypeAnnotation:
: *Type*

A.4 Functions

FunctionDeclaration: (Modified)
FunctionOverloads$_{opt}$ *FunctionImplementation*

FunctionOverloads:
FunctionOverload
FunctionOverloads *FunctionOverload*

FunctionOverload:
`function` *Identifier* *CallSignature* `;`

FunctionImplementation:
`function` *Identifier* *CallSignature* `{` *FunctionBody* `}`

A.5 Interfaces

InterfaceDeclaration:
`interface` *Identifier* *TypeParameters*$_{opt}$ *InterfaceExtendsClause*$_{opt}$ *ObjectType*

InterfaceExtendsClause:
`extends` *ClassOrInterfaceTypeList*

ClassOrInterfaceTypeList:
ClassOrInterfaceType
ClassOrInterfaceTypeList `,` *ClassOrInterfaceType*

ClassOrInterfaceType:
TypeReference

A.6 Classes

ClassDeclaration:
`class` *Identifier* *TypeParameters*$_{opt}$ *ClassHeritage* `{` *ClassBody* `}`

ClassHeritage:
ClassExtendsClause$_{opt}$ *ImplementsClause*$_{opt}$

ClassExtendsClause:
`extends` *ClassType*

ClassType:
TypeReference

ImplementsClause:
`implements` *ClassOrInterfaceTypeList*

ClassBody:
ClassElements$_{opt}$

ClassElements:
ClassElement
ClassElements *ClassElement*

ClassElement:
ConstructorDeclaration
MemberDeclaration
IndexSignature

ConstructorDeclaration:
ConstructorOverloads$_{opt}$ *ConstructorImplementation*

ConstructorOverloads:
ConstructorOverload
ConstructorOverloads *ConstructorOverload*

ConstructorOverload:
`constructor` `(` *ParameterList*$_{opt}$ `)` `;`

ConstructorImplementation:

`constructor` `(` *ParameterList*$_{opt}$ `)` `{` *FunctionBody* `}`

MemberDeclaration:

MemberVariableDeclaration
MemberFunctionDeclaration
MemberAccessorDeclaration

MemberVariableDeclaration:

PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *TypeAnnotation*$_{opt}$ *Initialiser*$_{opt}$ `;`

MemberFunctionDeclaration:

MemberFunctionOverloads$_{opt}$ *MemberFunctionImplementation*

MemberFunctionOverloads:

MemberFunctionOverload
MemberFunctionOverloads *MemberFunctionOverload*

MemberFunctionOverload:

PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `;`

MemberFunctionImplementation:

PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `{` *FunctionBody* `}`

MemberAccessorDeclaration:

PublicOrPrivate$_{opt}$ `static`$_{opt}$ *GetAccessor*
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *SetAccessor*

A.7 Enums

EnumDeclaration:

`enum` *Identifier* `{` *EnumBody*$_{opt}$ `}`

EnumBody:

ConstantEnumMemberList `,`$_{opt}$
ConstantEnumMemberList `,` *ComputedEnumMemberList* `,`$_{opt}$
ComputedEnumMemberList `,`$_{opt}$

ConstantEnumMemberList:

ConstantEnumMember
ConstantEnumMemberList `,` *ConstantEnumMember*

ConstantEnumMember:

PropertyName
PropertyName `=` *IntegerLiteral*

IntegerLiteral:
SignedInteger
HexIntegerLiteral

ComputedEnumMemberList:
ComputedEnumMember
ComputedEnumMemberList `,` *ComputedEnumMember*

ComputedEnumMember:
PropertyName `=` *AssignmentExpression*

A.8 Internal Modules

ModuleDeclaration:
`module` *IdentifierPath* `{` *ModuleBody* `}`

IdentifierPath:
Identifier
IdentifierPath `.` *Identifier*

ModuleBody:
ModuleElements$_{opt}$

ModuleElements:
ModuleElement
ModuleElements *ModuleElement*

ModuleElement:
Statement
`export`$_{opt}$ *VariableDeclaration*
`export`$_{opt}$ *FunctionDeclaration*
`export`$_{opt}$ *ClassDeclaration*
`export`$_{opt}$ *InterfaceDeclaration*
`export`$_{opt}$ *EnumDeclaration*
`export`$_{opt}$ *ModuleDeclaration*
`export`$_{opt}$ *ImportDeclaration*

ImportDeclaration:
`import` *Identifier* `=` *EntityName* `;`

EntityName:
Identifier
ModuleName `.` *Identifier*

A.9 Programs and External Modules

SourceFile:
    ImplementationSourceFile
    DeclarationSourceFile

ImplementationSourceFile:
    ImplementationElements$_{opt}$

ImplementationElements:
    ImplementationElement
    ImplementationElements *ImplementationElement*

ImplementationElement:
    ModuleElement
    ExportAssignment
    `export`$_{opt}$ *ExternalImportDeclaration*
    `export`$_{opt}$ *AmbientDeclaration*

DeclarationSourceFile:
    DeclarationElements$_{opt}$

DeclarationElements:
    DeclarationElement
    DeclarationElements *DeclarationElement*

DeclarationElement:
    ExportAssignment
    `export`$_{opt}$ *InterfaceDeclaration*
    `export`$_{opt}$ *ImportDeclaration*
    `export`$_{opt}$ *ExternalImportDeclaration*
    `export`$_{opt}$ *AmbientDeclaration*

ExternalImportDeclaration:
    `import` *Identifier* `=` *ExternalModuleReference* `;`

ExternalModuleReference:
    `require` `(` *StringLiteral* `)`

ExportAssignment:
    `export` `=` *Identifier* `;`

A.10 Ambients

AmbientDeclaration:
`declare` *AmbientVariableDeclaration*
`declare` *AmbientFunctionDeclaration*
`declare` *AmbientClassDeclaration*
`declare` *AmbientEnumDeclaration*
`declare` *AmbientModuleDeclaration*
`declare` *AmbientExternalModuleDeclaration*

AmbientVariableDeclaration:
`var` *Identifier* *TypeAnnotation*$_{opt}$ `;`

AmbientFunctionDeclaration:
`function` *Identifier* *CallSignature* `;`

AmbientClassDeclaration:
`class` *Identifier* *TypeParameters*$_{opt}$ *ClassHeritage* `{` *AmbientClassBody* `}`

AmbientClassBody:
AmbientClassBodyElements$_{opt}$

AmbientClassBodyElements:
AmbientClassBodyElement
AmbientClassBodyElements *AmbientClassBodyElement*

AmbientClassBodyElement:
AmbientConstructorDeclaration
AmbientMemberDeclaration
IndexSignature

AmbientConstructorDeclaration:
`constructor` `(` *ParameterList*$_{opt}$ `)` `;`

AmbientMemberDeclaration:
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *TypeAnnotation*$_{opt}$ `;`
PublicOrPrivate$_{opt}$ `static`$_{opt}$ *PropertyName* *CallSignature* `;`

AmbientEnumDeclaration:
`enum` *Identifier* `{` *AmbientEnumBody*$_{opt}$ `}`

AmbientEnumBody:
AmbientEnumMemberList `,`$_{opt}$

AmbientEnumMemberList:
    AmbientEnumMember
    AmbientEnumMemberList `,` *AmbientEnumMember*

AmbientEnumMember:
    PropertyName
    PropertyName `=` *IntegerLiteral*

AmbientModuleDeclaration:
    `module` *IdentifierPath* `{` *AmbientModuleBody* `}`

AmbientModuleBody:
    AmbientModuleElements$_{opt}$

AmbientModuleElements:
    AmbientModuleElement
    AmbientModuleElements *AmbientModuleElement*

AmbientModuleElement:
    `export`$_{opt}$ *AmbientVariableDeclaration*
    `export`$_{opt}$ *AmbientFunctionDeclaration*
    `export`$_{opt}$ *AmbientClassDeclaration*
    `export`$_{opt}$ *InterfaceDeclaration*
    `export`$_{opt}$ *AmbientEnumDeclaration*
    `export`$_{opt}$ *AmbientModuleDeclaration*
    `export`$_{opt}$ *ImportDeclaration*

AmbientExternalModuleDeclaration:
    `module` *StringLiteral* `{` *AmbientExternalModuleBody* `}`

AmbientExternalModuleBody:
    AmbientExternalModuleElements$_{opt}$

AmbientExternalModuleElements:
    AmbientExternalModuleElement
    AmbientExternalModuleElements *AmbientExternalModuleElement*

AmbientExternalModuleElement:
    AmbientModuleElement
    ExportAssignment
    `export`$_{opt}$ *ExternalImportDeclaration*