TypeScript

Language Specification

Version 0.9

February 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.

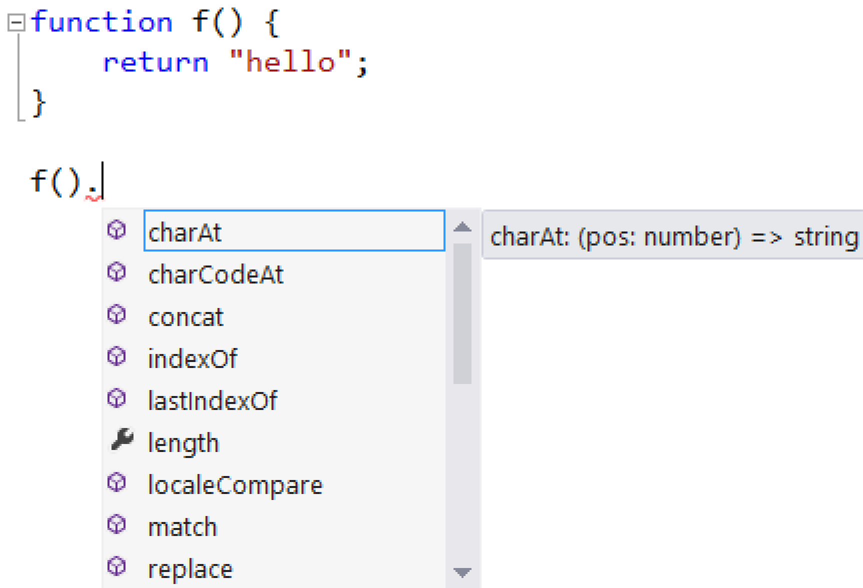
The TypeScript type system enables programmers to express limits on the capabilities of JavaScript objects, and to use tools that enforce these limits. To minimize the number of annotations needed for tools to become useful, the TypeScript type system makes extensive use of type inference. For example, 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 3.6.2 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.

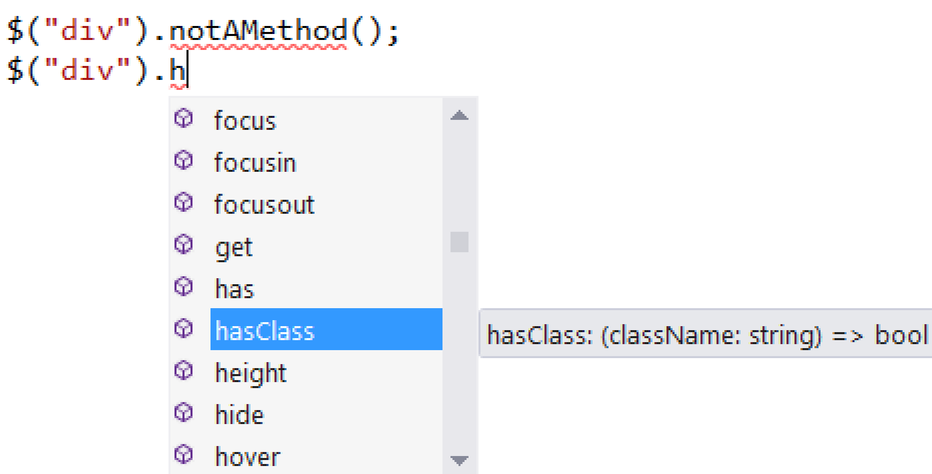
```
var f: { (): string; };
var sameType: () => string = f;     // Ok
var nope: () => number = sameType;  // Error: type mismatch
```

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 3.7 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` *Arguments*
   `super .` *Identifier*

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

2.2 Namespace and Type Names

TypeScript supports named types that can be organized in hierarchical namespaces. Namespaces are introduced by module declarations and named types are introduced by class and interface 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 type. Type and namespace names are in separate declaration spaces and it is therefore possible for a type and a namespace to have the same name, as in the example above.

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

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

introduces a class instance 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 has a declaration space for global variables (including functions, modules, and class constructor functions), a declaration space for global types (classes and interfaces), and a declaration space for global namespaces (containers of types).
- Each module body has a declaration space for local variables (including functions, modules, and class constructor functions), a declaration space for local types (classes and interfaces), and a declaration space for local namespaces (containers of types). Every declaration (whether local or exported) in a module contributes to one or more of these declaration spaces.
- Each internal or external module has a declaration space for exported members, a declaration space for exported 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 interface type 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 object type literal has a declaration space for its members.
- Each class declaration has a declaration space for instance members, a declaration space for static members, and a declaration space for type parameters.
- 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.

Top-level declarations in a non-module source file belong to the ***global module***. Top-level declarations in a module source file 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 instance type) to the containing module. An interface declaration contributes a type name to the containing module. Any other declaration contributes a member name to the declaration space to which it belongs.

The ***root module*** of an entity declared in a module is the outermost module within which the entity is reachable. Specifically, the root module of an entity *M* in a parent module *P* is determined as follows:

- If *P* is the global module or an external module, *M*'s root module is *P*.
- If *M* is not exported, *M*'s root module is *P*.
- If *M* is exported, *M*'s root module is the root module of *P*.

Interfaces and internal modules are "open ended," and interface and internal module declarations with the same qualified name relative to a common root are automatically merged. For further details, see section 9.3.

Namespace, type, and member names exist in separate declaration spaces. Furthermore, module declarations that contain only type or module declarations 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 type 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 (i.e. a module containing statements or member declarations) 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 global variable, function, class, module, or interface is the entire program text.
- The scope of a non-exported variable, function, class, module, or interface declared within a module declaration is the body of that module declaration.
- The scope of an exported variable, function, class, module, or interface 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 an exported variable, function, class, module, or interface declared in an external module is the body of that module.
- 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 3.5.2), only classes, interfaces, and type parameters are considered and other entities in scope are ignored.

When an identifier is resolved as a *ModuleName* (section 3.5.2), 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 9.3) and classes, functions, variables, and parameters are considered and other entities in scope are ignored.

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

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. The `number`, `bool`, `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'.

NOTE: The TypeScript compiler currently implements an experimental form of enum types. We expect the final language to support enum types, but not in the form they are currently implemented.

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 no 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.

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 3.7) 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 `bool` 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 3.7) 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: bool;            // Explicitly typed
var yes = true;         // Same as yes: bool = true
var no = false;         // Same as no: bool = 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 3.7) 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 method 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
```

3.3 Object Types

The object types include all class, interface, and array 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 Class and Interface Types

Class and interface types are named types that are introduced through class declarations (section 8.1) and interface declarations (section 7.1). Class and interface types may have type parameters and are then called generic types. Class and interface types are referenced through type references (section 3.5.2) that specify the name of the class or interface and, if applicable, the type arguments to be substituted for the type parameters of the class or interface.

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

3.3.2 Array Types

Array types represent JavaScript arrays. Array types are type references (section 3.5.2) created from the generic interface type 'Array' in the global module. Array type literals (section 3.5.3) 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 3.5.3).
- Object type literals (section 3.6).
- Object literals (section 4.5).
- Function expressions (section 4.9) and function declarations (6.1).
- Constructor function types created by class declarations (section 8.2.4).
- Module instance types created by module declarations (section 9.2).

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 an object of the given type. Property names are unique within their type.
- ***Call signatures***, which define the possible parameter lists and the return type associated with applying a call operation to an object of the given type.
- ***Construct signatures***, which define the possible parameter lists and the return type associated with applying the new operator to an object of the given type.
- ***Index signatures***, which define the index expression types and the return type associated with applying an index operation to an object of 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 8.2.1, and private properties match only themselves in subtype and assignment compatibility checks, as described in section 3.7.
- 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 3.7.3.

For purposes of determining type relationships (section 3.7) 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:

```
Object o = { x: 10, y: 20 };         // Ok
Function f = (x: number) => x * x;   // Ok
Object err = { 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 can be instantiated with many different actual type arguments, type parameters have certain restrictions compared to other types. In particular, a type parameter cannot be used as a base class, base interface, or type parameter constraint.

For purposes of determining type relationships (section 3.7), 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.

3.4.1 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 *TypeParameterConstraint*$_{opt}$

TypeParameterConstraint:
  `extends` *Type*

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 Specifying Types

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

Type:
  PredefinedType
  TypeReference
  TypeLiteral

3.5.1 Predefined Types

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

PredefinedType:
```
any
number
bool
string
void
```

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

3.5.2 Type References

A type reference references a class, interface, or type parameter through its name and an optional 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 class, interface, or type parameter. A reference to a generic class or interface may optionally 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 class or interface member *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.

3.5.2.1 Generic Type Arguments

A type reference to a generic type may optionally include a list of type arguments enclosed in angle brackets (< and >) and separated by commas.

TypeArguments:
  `<` *TypeArgumentList* `>`

TypeArgumentList:
  TypeArgument
  TypeArgumentList `,` *TypeArgument*

TypeArgument:
  Type

A type reference that includes a type argument list 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 3.7.3) the constraint of the corresponding type parameter. Omitting the type argument list in a reference to a generic type corresponds to specifying type Any as the type argument for every type parameter. 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;                     // Ok, equivalent to G<any, any>
```

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 class or interface 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 };
};
```

3.5.3 Type Literals

Type literals compose other types into new anonymous types.

TypeLiteral:
    ObjectType
    ArrayType
    FunctionType
    ConstructorType

ArrayType:
    Type `[` `]`

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 3.6. 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.

3.6 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* }

TypeBody:
  TypeMemberList$_{opt}$
  TypeMemberList ;

TypeMemberList:
  TypeMember
  TypeMemberList ; *TypeMember*

TypeMember:
  PropertySignature
  CallSignature
  ConstructSignature
  IndexSignature
  FunctionSignature

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

3.6.1 Property Signatures

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

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

The *Identifier* of a property signature must be unique within its containing type. If the identifier 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.

3.6.2 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 3.4.1) 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 function signatures (section 3.6.5), 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 3.7.1) or differ only by their return types.

3.6.2.1 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.

3.6.2.2 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* $TypeAnnotation_{opt}$

PublicOrPrivate:
`public`
`private`

OptionalParameterList:
OptionalParameter
OptionalParameterList `,` *OptionalParameter*

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

RestParameter:
`...` *RequiredParameter*

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[]`.

A parameter can be marked optional by following its name with a question mark (`?`). Additionally, in the signature of a function implementation (section 6.3), a parameter can be marked optional by following it with an initializer.

TODO: Rest parameters.

3.6.2.3 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.

3.6.3 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 3.7.1) or differ only by their return types.

3.6.4 Index Signatures

An index signature defines the parameter and return type associated with applying an indexing operation (section 4.10) to an instance of the containing type.

> *IndexSignature:*
>   `[` *RequiredParameter* `]` *TypeAnnotation*$_{opt}$

The type of an index signature's parameter must be the Number or String primitive type. A type may overload indexing operations by defining index signatures with both parameter types.

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

3.6.5 Function Signatures

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

> *FunctionSignature:*
>   *Identifier* `?`$_{opt}$ *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. Therefore, only *FunctionSignatures* occurring in *ObjectTypes* are permitted to include a question mark after their identifier.

A function signature of the form

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

is equivalent to the property declaration

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

A literal type may ***overload*** a function by declaring multiple function 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 function signatures correspond to a declaration of a single property with a type composed from an equivalent set of call signatures. Specifically

> *Identifier* `<` *TypeParamList*$_1$ `>` `(` *ParamList*$_1$ `)` `:` *ReturnType*$_1$ `;`
> *Identifier* `<` *TypeParamList*$_2$ `>` `(` *ParamList*$_2$ `)` `:` *ReturnType*$_2$ `;`
> …
> *Identifier* `<` *TypeParamList*$_n$ `>` `(` *ParamList*$_n$ `)` `:` *ReturnType*$_n$ `;`

is equivalent to

> *Identifier* `:` `{`
> `<` *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;         // Function 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.

3.7 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.

Finally, all type parameters appear to have the members of their constraint (or the 'Object' interface if they have no constraint), but no other members.

3.7.1 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 with identical types,
- they are identical call signatures,
- they are identical construct signatures, or
- they are index signatures with identical parameter and return 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 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.

Named types (such as classes or 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 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 the same *S* and *T* (where same means originating in the same declaration). For example, consider the relationship between ‘A’ above and ‘B’ below:

```
interface B { next: C; }

interface C { next: D; }

interface D { next: B; }
```

To compare ‘A’ and ‘B’, 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 assumed by definition to be 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 7.3.

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.

3.7.2 Subtypes and Supertypes

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

- if *S* is the primitive type Number, Boolean, or String, *S’* is the global interface type ‘Number’, ‘Boolean’, or ‘String’,
- if *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'* 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 call, construct or index signature and *S'* contains a call, construct or index 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, construct or index),
        - 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*.

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

3.7.3 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

- if *S* is the primitive type Number, Boolean, or String, *S'* is the global interface type 'Number', 'Boolean', or 'String',
- if *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'* 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 call, construct or index signature and *S'* contains a call, construct or index 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, construct or index),

- 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*.

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

Note that the assignment compatibility and subtyping rules differ only in that the Any type is assignable to, but not a subtype of, all types.

The rules above mean 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
```

3.8 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[]
```

3.9 The Best Common Type

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

Given a set of expressions { $e_1, e_2, ..., e_n$ } of types { $T_1, T_2, ..., T_n$ }, the best common type of those expressions is the one T_x in the set that is a supertype of every T_n. It is possible that no such type exists, or more than one such type exists, in which case the given set of expressions has no best common 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 the constructor function type of the containing class.
- 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, 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 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 RegExp interface type.

4.5 Object Literals

The type of an object literal is an object type with the set of properties specified in the object literal. For each property assignment *Name* `:` *Expr* in the object literal, the type of the resulting property is determined as follows:

- If the object literal is contextually typed (section 4.18) and the contextual type contains a property *p* with the given *Name*, then *Expr* is contextually typed by the type of *p*. The type of *Expr* must be assignable to the type of *p* and the resulting property has that type.
- Otherwise, the type of the resulting property is the widened form (section 3.8) of the type of *Expr*.

It is a compile-time error to specify multiple values for the same property.

4.6 Array Literals

The type of an array literal is determined as follows:

- If the array literal is contextually typed (section 4.18) by an array type *E*`[]`, then each element expression is contextually typed by *E*. The type of each element expression must be assignable to *E*, and the type of the array literal is *E*`[]`.
- Otherwise, if the array literal is empty, the type of the array literal is `any[]`.

- Otherwise, if a best common type of each of the element expressions can be determined, the type of the array literal is *E*`[]`, where *E* is that best common type.
- Otherwise, no type can be determined for the array literal and a compile-time error occurs.

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` *Arguments*
`super` `.` *Identifier*

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.

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 instance member functions from derived classes.

A super property access is permitted only in a constructor, instance member function, or instance member accessor of a derived class and must specify a public instance member function of the base class. 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 instance member functions from derived class instance 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 6.4).

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 would have a meaningless `this`.

In the example

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

the use of an arrow function expression causes the callback to have the same `this` as the surrounding start function. 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:

```
var messenger = {
    message: "Hello World",
    start: function() {
        var _this = this;
        setTimeout(function() { alert(_this.message); }, 3000);
    }
};
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 and return 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. Furthermore, in a property access, an object (including Number, Boolean, or String) appears to have additional properties and indexers 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 the 'Object' global interface, as described in section 3.4.

A dot notation property access of the form

ObjExpr `.` *Identifier*

where *ObjExpr* is an expression and *Identifier* is an identifier, 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 the Any type, any *Identifier* is permitted and the type of the result is of type Any.
- Otherwise, if *ObjExpr* is of a primitive type, object type, or type parameter type and *Identifier* denotes a property member in that type, the result 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 *ObjExpr* is of type Any and *IndexExpr* is of type Any or the Number or String primitive type, the result is of type Any.
- Otherwise, if *ObjExpr* is of a primitive type, object type, or type parameter type that has one or more index signatures, the property access is processed in the same manner as a function call, but using the index 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.
- Otherwise, the property access is invalid and a compile-time error occurs.

The global interface type 'Object' by default defines an indexer '[s: string]: any'. Thus, objects of any type can be indexed by string expressions for computed access to their properties.

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 best common type (section 3.9) of the return types of the remaining signatures in the set. If no such best common type exists, the function call is ambiguous and a compile-time error occurs.

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 3.7.3) that parameter type.

Given an argument list A with a set of argument types { A_1, A_2, ..., A_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 each argument, the conversion from A_X to Q_X is not better than the conversion from A_X to P_X, and
- for at least one argument, the conversion from A_X to P_X is better than the conversion from A_X to Q_X.

Given a conversion C_1 from a type S to a type T_1 and a conversion C_2 from a type S to a type T_2, the ***better conversion*** of the two conversions is determined as follows:

- If T_1 and T_2 are identical types, neither conversion is better.
- If S is identical to T_1, C_1 is the better conversion.
- If S is identical to T_2, C_2 is the better conversion.
- If T_1 is a subtype of T_2, C_1 is the better conversion.
- If T_2 is a subtype of T_1, C_2 is the better conversion.
- Otherwise, neither conversion is better.

4.12.2 Type Argument Inference

Given a signature < T_1 , T_2 , ... , T_n > (p_1 : P_1 , p_2 : P_2 , ... , 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 , ... , 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 3.9) of a set of candidate types, or the type Any if no best common type exists or the set of candidates is empty. 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 ***typed for inference*** 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.

An expression e is typed for inference by a type T as follows:

- If e is an *ObjectLiteral* and T is an object type, then for each property assignment *Name* **:** *Expr* in the object literal, if T contains a property p with the given *Name*, then *Expr* is typed for inference by the type of p. Otherwise, *Expr* processed as a regular expression with no contextual type. The type of the resulting property is the widened form (section 3.8) of the type of *Expr*.
- If e is an *ArrayLiteral* and T is an array type, then each element expression is typed for inference by the element type of T. If the array literal is not empty and if a best common type, C, of the element expressions can be determined, the resulting type of the array literal is C[]. Otherwise, the resulting type of the array literal is any[].

- If *e* is a *FunctionExpression* or *ArrowFunctionExpression* with no parameter or return type annotations and *T* is an object type with exactly one call signature, then the function expression is processed as if it had explicitly specified type annotations as they exist in *T*, but with inferred type arguments substituted for their respective type parameters. Any inferred type arguments referenced in this manner become ***fixed*** and no further candidate inferences are made for them.
- Otherwise, *e* is processed as a regular expression with no contextual type.

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 array types, candidate inferences are made by relating the element type of *S* to the element type of *T*.
- 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, construct or index 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*.

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 method '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 method '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' method *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 or the Number primitive 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 or the Number primitive type. 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 at least one of the operands to be of type Any or the String primitive type. 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 3.9) of the two operand types, or of type Any if no best common type can be determined.

	Any	Boolean	Number	String	Object
Any	Any	Any	Any	Any	Any
Boolean	Any	Boolean	Any	Any	Any
Number	Any	Any	Number	Any	Any
String	Any	Any	Any	String	Any
Object	Any	Any	Any	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 3.9) of the two expressions.

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 3.7.3) 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 array type, *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*

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 3.8) 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.

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
```

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 optional 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` *FunctionSignature* `;`
>
> *FunctionImplementation:*
>   `function` *FunctionSignature* `{` *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.

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 3.8) of the best common type (section 3.9) of the return statement expressions in the function body, ignoring return statements with no expressions. A compile-time error occurs if no best common type can be determined.

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.

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

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 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 3.7.1).

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 object type.

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

InterfaceExtendsClause:
  `extends` *ClassOrInterfaceTypeList*

ClassOrInterfaceTypeList:
  ClassOrInterfaceType
  ClassOrInterfaceTypeList `,` *ClassOrInterfaceType*

ClassOrInterfaceType:
  TypeReference

An interface may optionally have type parameters (section 3.4.1) 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*** (7.3).

An interface can inherit from zero or more ***base object types***, which can be class or interface types. The names of the base object types, if any, are specified in the *InterfaceExtendsClause*.

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

- A property declaration hides a public base object type property with the same name.

- A call signature declaration hides a base object type call signature that is identical when return types are ignored.
- A construct signature declaration hides a base object type construct signature that is identical when return types are ignored.
- An index signature declaration hides a base object type index signature with an identical parameter type.

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 object type that originates in the same declaration. In other words an interface cannot, directly or indirectly, be a base object 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 3.7.1).
- The declared interface must be a subtype (section 3.7.2) of each of its base object types.

An interface is permitted to inherit identical members from multiple base object 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 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 3.7).

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.

7.3 Generic Interfaces

Interface declarations may include type parameters and are then called generic interface declarations. Generic interface declarations allow multiple distinct types to be created from a single shared "template". A reference to a generic interface must include type arguments that are substituted in place of the interface's type parameters to form the resulting type. Because TypeScript has a structural type system, a type created from a reference to a generic interface is indistinguishable from an equivalent manually written expansion. For example, given the declaration

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

the types

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

and

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

are indistinguishable.

Generic interfaces 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 interface $G < T_1 , T_2 , \ldots , T_n >$ 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'.

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:

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

If such a test 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.

TODO: Update this chapter for generics.

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 instance type*** and a ***constructor function***, both with the name given by *Identifier*, in the containing module. The class instance 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 instance type.

The following example introduces both a type called 'Point' (the class instance 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 (section 3.3), the class instance type 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 cannot, directly or indirectly, be a base class of itself.
- The class instance type declared by the class declaration must be a subtype of the base class instance type and each of the interfaces 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 only situation in which the second or third 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

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.

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

Instance members are members of the class instance type. Within constructors, instance member functions, and instance member accessors, the type of `this` is the class instance type.

Static members are declared using the `static` modifier and are members of the constructor function type. Within static member functions and static member accessors, the type of `this` is the constructor function type.

TODO: Type parameters cannot be referenced in static member declarations.

8.2.1 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, whereas 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.

It is not possible to specify the accessibility of statics—they are effectively always public.

TODO: Allow accessibility modifiers on statics.

8.2.2 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 3.7.2) 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.

8.2.3 Class Instance Types

The named type declared by a class declaration is called the class instance type. Within the constructor and member functions of a class, the type of `this` is the class instance type. The class instance 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: bool) { return false; }
}
```

the class instance type of 'A' is

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

and the class instance type of 'B' is

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

Note that static declarations in a class do not contribute to the class 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'.

TODO: In a generic class declaration, the instance type is the type reference formed by referencing the class type itself with its own type parameters as type arguments.

8.2.4 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, returning the class instance type.
- 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, but returning the class instance type.

- If the class contains a constructor declaration with no overloads, a construct signature with the parameter list of the constructor implementation, returning the class instance type.
- If the class contains a constructor declaration with overloads, a set of construct signatures with the parameter lists of the overloads, returning the class instance type.
- 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.

TODO: In a generic class declaration, the constructor function type contains a single construct signature with type parameters corresponding to those of the class type. For example, a class Dictionary<K, V> with a parameterless constructor would have the construct signature new <K, V>( ): Dictionary<K, V>.

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 are present, they must specify expressions of types that are assignable to the class instance type.

TODO: Update for generics.

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 8.2.3), 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 8.2.4), 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 8.2.2, 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' of the class instance type. It is an error to explicitly declare a static member with that name.

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`$_{opt}$ *VariableDeclaration* `;`

The type associated with a member variable declaration is determined in the same manner as an ordinary variable declaration (see section 5.1). Member variable initializer expressions are not permitted to access `this`.

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.

A static member variable declaration introduces a property in the constructor function type and optionally initializes a property on the constructor function object.

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: bool;
    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`$_{opt}$ *FunctionSignature* `;`

MemberFunctionImplementation:
    PublicOrPrivate$_{opt}$ `static`$_{opt}$ *FunctionSignature* `{` *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, `this` is of the constructor function type.

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`$_{opt}$ *GetAccessorSignature* `{` *FunctionBody* `}`
    PublicOrPrivate$_{opt}$ `static`$_{opt}$ *SetAccessorSignature* `{` *FunctionBody* `}`

GetAccessorSignature:
    `get` *Identifier* `(` `)` *TypeAnnotation*$_{opt}$

SetAccessorSignature:
    `set` *Identifier* `(` *RequiredParameter* `)`

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

If a class declares both a get and a set accessor for a particular member name and kind (instance or static), and if the set accessor omits the type annotation for its parameter, the parameter type is inferred to be the return type of the get accessor (which in turn might be inferred from the get accessor body).

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, `this` is of the constructor function type.

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>
    }
    <MemberAndStaticStatements>
    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 6.4.

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.

MemberAndStaticStatements is a sequence of statements, one for each static member variable declaration with an initializer, member function declaration, or member accessor declaration, in the order they are declared.

A static variable declaration with an initializer generates a statement 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.

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 6.4.

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.

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>
    }
    <MemberAndStaticStatements>
    return <ClassName>;
})(<BaseClassName>);
```

In addition, the '__extends' method 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 statics), 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, a member function, or a 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.

9 Programs and Modules

TypeScript implements 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.

9.1 Programs

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 ambient declarations only.

SourceFile:
    ImplementationSourceFile
    DeclarationSourceFile

ImplementationSourceFile:
    ModuleBody

ModuleBody:
    ModuleElements$_{opt}$

ModuleElements:
    ModuleElement
    ModuleElements ModuleElement

ModuleElement:
    Statement
    FunctionDeclaration
    ClassDeclaration
    InterfaceDeclaration
    ModuleDeclaration
    ImportDeclaration
    ExportDeclaration
    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 *SourceElements* 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 *ClassDeclaration* (section 8.1), *InterfaceDeclaration* (section 7.1), *ModuleDeclaration* (section 9.2), *ImportDeclaration* (section 9.2.2), *ExportDeclaration* (section 9.2.1), and *AmbientDeclaration* productions (section 10.1).

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. Declaration source files are described in further detail in section 10.2.

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 used but generally it is the case that non-circularly dependent modules are automatically loaded and initialized in the correct order.

9.1.1 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 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 import declaration that specifies a relative external module name (section 9.4.1) 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 import declaration that specifies a top-level external module name (section 9.4.1) 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.

9.2 Module Declarations

TODO: Describe the dual nature of modules, i.e. both object instances and type containers.

A module is a body of statements and declarations that create and initialize a singleton module instance. Members exported from a module become properties on the module instance. The body of a module corresponds to a function that is executed once, thereby providing a mechanism for maintaining local state with assured isolation.

ModuleDeclaration:
  `module` *IdentifierPath* `{` *ModuleBody* `}`

IdentifierPath:
  Identifier
  IdentifierPath `.` *Identifier*

TypeScript supports two types of modules.

Internal modules (section 9.2.2) are local or exported members of other modules (including the global module and external modules). Internal modules are declared using *ModuleDeclarations* that specify their name and body. A name path with more than one identifier is equivalent to a series of nested internal module declarations.

External modules (section 9.4) are separately loaded bodies of code referenced using ***external module names***. An external module is written as a separate source file that contains at least one import or export declaration. In addition, external modules can be declared using *AmbientModuleDeclarations* in the global module that directly specify the external module names as string literals. This is described further in section 10.1.4.

9.2.1 Export Declarations

TODO: Reference export assignments in external modules.

An export declaration declares an externally accessible module member.

ExportDeclaration:
  `export` *VariableStatement*
  `export` *FunctionDeclaration*
  `export` *ClassDeclaration*
  `export` *InterfaceDeclaration*
  `export` *ModuleDeclaration*
  `export` *AmbientDeclaration*
  ExportAssignment

ExportAssignment:
  `export` `=` *Identifier* `;`

An export declaration is simply a regular declaration prefixed with the keyword `export`.

Exported variable, function, class, and module 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 internal module declaration.

An exported member depends on a (possibly empty) set of named types (section 3.5.2). 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 its type.
- A function directly depends on its function type.
- A class directly depends on its constructor function type and its class instance type.
- An interface directly depends on the type it declares.
- A module directly depends on its module instance type.
- An object type (section 3.3) directly depends on the types of each of its public properties and the parameter and return types of each of its call, construct, and index signatures.

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.

9.2.2 Import Declarations

Import declarations are used to import internal or external modules and create local aliases by which they may be referenced.

ImportDeclaration:
  `import` *Identifier* `=` *ModuleReference* `;`

ModuleReference:
  ExternalModuleReference
  ModuleName

ExternalModuleReference:
  `module` `(` *StringLiteral* `)`

An import declaration introduces a local identifier that references a given module. The local identifier is itself classified as a module and behaves exactly as such. It is not possible to export a module identifier declared in an import declaration.

The string literal specified in an *ExternalModuleReference* is interpreted as an external module name (section 9.4.1). Import declarations in external modules can specify either *ExternalModuleReferences* or *ModuleNames*, but import declarations in internal modules can only specify *ModuleNames*.

TODO: Specify the exact restrictions on import declarations referencing other import declarations. We minimally want to disallow circular references.

9.2.3 Module Identifiers

A module declaration or an import declaration declares an identifier that references a module. A module identifier may be used as a *PrimaryExpression* (section 4.3), in which case it denotes the singleton module instance or as a *ModuleName* (section 3.5.2), in which case it denotes a container of module and type names. For example:

```
module M {
    export interface P { x: number; y: number; }
    export var a = 1;
}

var p: M.P;              // Used as ModuleName
var m = M;               // Used as PrimaryExpression
var x1 = M.a;            // 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.

9.3 Internal Modules

TODO: Describe instantiated vs. non-instantiated modules: An instantiated module is one for which a module instance is created. A non-instantiated module contains only types and other non-instantiated modules.

TODO: A module declaration introduces a namespace in the containing module and, if the module is instantiated, a member in the containing module.

TODO: An instantiated module may have the same name as a function or class provided the declaration of the internal module textually follows that of the function or class and is located in the same source file. In the case of a function this provides a way of adding properties to the function object. In the case of a class, this is an alternative way to write static members.

Internal modules are declared using *ModuleDeclarations*. An internal module is a local or exported member of another module (which might be the global module or an external module).

The following is an example of an internal module:

```
module Utils {
    var toString = Object.prototype.toString;
    export function isFunction(obj) {
        return toString.call(obj) == "[object Function]";
    }
    export function isNumber(obj) {
        return toString.call(obj) == "[object Function]";
    }
}
```

The JavaScript code emitted for this module is:

```
var Utils;
(function(Utils) {
    var toString = Object.prototype.toString;
    function isFunction(obj) {
        return toString.call(obj) == "[object Function]";
    }
    Utils.isFunction = isFunction;
    function isNumber(obj) {
        return toString.call(obj) == "[object Function]";
    }
    Utils.isNumber = isNumber;
})(Utils||(Utils={}));
```

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 function doesn't create and return a 'Utils' module instance, but rather it extends the extisting instance (which may have just been created in the function call). This ensures that internal modules can extend each other.

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;
    };
}
```

If the *IdentifierPath* of an internal module specifies more than one identifier, it corresponds to a series of nested module declarations where all but the first identifier 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;
        }
    }
}
```

Unlike external modules, internal modules do not have to be imported before they can be referenced. As long as an internal module's identifier is in scope it is possible to reference the internal module directly. For example:

```
module A.B.C {
    export var x = 1;
}

module M {
    var y = A.B.C.x;
}
```

Internal module aliases created by import declarations are evaluated lazily. In intuitive terms, internal module aliases behave as if the module name paths they represent are substituted in their place. This means that import declarations can contain forward references as long as expressions that use those import declarations aren't evaluated until after the module instances are created. In the example

```
module A.B.C {
    import XYZ = X.Y.Z;
    export function ping(x: number) {
        if (x > 0) XYZ.pong(x - 1);
    }
}

module X.Y.Z {
    import ABC = A.B.C;
    export function pong(x: number) {
        if (x > 0) ABC.ping(x - 1);
    }
}
```

the references to 'XYZ.pong' and 'ABC.ping' behave exactly as if 'X.Y.Z.pong' and 'A.B.C.ping' had been written. Had the import declarations instead been variable declarations, the example would fail because module 'A.B.C' would attempt to evaluate the expression 'X.Y.Z' prior to module X.Y.Z's creation.

9.4 External Modules

TODO: Describe export assignments and the ability to export non-modules.

External modules are separately loaded bodies of code referenced using external module names. External modules are written as separate source files that contain at least one *ImportDeclaration* or *ExportDeclaration*.

Below is an example of two external modules written in separate source files.

File main.ts:

```
import log = module("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 9.4.2), typically used by server frameworks such as node.js, and the Asynchronous Module Definition (AMD) pattern (section 9.4.3), 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.

9.4.1 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 *AmbientModuleDeclaration* (section 10.1.4) 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 *AmbientModuleDeclaration* (section 10.1.4) 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.

9.4.2 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 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 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 = module("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 = module("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 };
```

9.4.3 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.

9.5 Code Generation

9.5.1 Internal Modules

An internal module generates JavaScript code that is equivalent to the following:

```
var <ModuleName>;
(function(<ModuleName>) {
    <ModuleStatements>
})(<ModuleName>||(<ModuleName>={}));
```

ModuleName is the name of the module.

ModuleStatements is the code generated for the statements in the module body.

TODO: Finish this section.

10 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, 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.

In implementation source files (files with a '.ts' extension), ambient declarations are written using the `declare` keyword. In declaration source files (files with a '.d.ts' extension), a `declare` keyword is implicitly present on every declaration.

TODO: Finish this chapter.

10.1 Ambient Declarations

Ambient declarations can declare variables, functions, classes, or modules.

> *AmbientDeclaration:*
> `declare` *AmbientVariableDeclaration*
> `declare` *AmbientFunctionDeclaration*
> `declare` *AmbientClassDeclaration*
> `declare` *AmbientModuleDeclaration*

10.1.1 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.

10.1.2 Ambient Function Declarations

An ambient function declaration introduces a function in the containing declaration space.

> *AmbientFunctionDeclaration:*
> `function` *FunctionSignature* `;`

Ambient functions may be overloaded by specifying multiple ambient function declarations with the same name, but no two declarations for the same name may have the same number of parameters and identical parameter types in the respective positions.

Ambient function declarations cannot specify a function bodies and do not permit default parameter values.

10.1.3 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

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

AmbientMemberDeclaration:
  PublicOrPrivate$_{opt}$ `static`$_{opt}$ *Identifier* *TypeAnnotation*$_{opt}$ `;`
  PublicOrPrivate$_{opt}$ `static`$_{opt}$ *FunctionSignature* `;`

10.1.4 Ambient Module Declarations

An ambient module declaration declares an internal or external module.

AmbientModuleDeclaration:
  `module` *AmbientModuleIdentification* `{` *AmbientModuleBody* `}`

AmbientModuleIdentification:
  IdentifierPath
  StringLiteral

AmbientModuleBody:
  AmbientElements$_{opt}$

AmbientElements:
  AmbientElement
  AmbientElements *AmbientElement*

AmbientElement:
`export`$_{opt}$ *AmbientVariableDeclaration*
`export`$_{opt}$ *AmbientFunctionDeclaration*
`export`$_{opt}$ *AmbientClassDeclaration*
`export`$_{opt}$ *InterfaceDeclaration*
`export`$_{opt}$ *AmbientModuleDeclaration*
ImportDeclaration
ExportAssignment

An *AmbientModuleIdentification* with a *StringLiteral* 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.

An *ImportDeclaration* in an *AmbientModuleDeclaration* for an external module may reference other external modules only through top-level external module names. Relative external module names are not permitted.

An *AmbientElement* declared an internal or external module must specify an export modifier (unless it is an *ImportDeclaration*). Conversely, an *AmbientElement* declared in the global module cannot specify an export modifier.

10.2 Declaration Source Files

Declaration source files (files with a '.d.ts' extension) are restricted to contain ambient declarations only. Declaration source files can be used to declare the static type information associated with existing JavaScript code in an adjunct manner.

DeclarationSourceFile:
AmbientModuleBody

A *DeclarationSourceFile* that contains no import or export declarations contributes to the global module.

A *DeclarationSourceFile* that contains at least one import or export declaration is considered a separate external module.