3 Types

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

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

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

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

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

```
var x: number;
```

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

```
var x = 1;
```

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