Type arguments correspond one-to-one with type parameters of the generic type or function being referenced. A type argument list is required to specify exactly one type argument for each corresponding type parameter, and each type argument is required to ***satisfy*** the constraint of its corresponding type parameter. A type argument satisfies a type parameter constraint if the type argument is assignable to (section 3.8.4) the constraint type once type arguments are substituted for type parameters.

Given the declaration

```
interface G<T extends U, U extends Array<V>, V extends Function> { }
```

a type reference of the form 'G<A, B, C>' requires 'A' to be assignable to 'B', 'B' to be assignable to 'Array<C>', and 'C' to be assignable to 'Function'.

The process of substituting type arguments for type parameters in a generic type or generic signature is known as ***instantiating*** the generic type or signature. Instantiation of a generic type or signature can fail if the supplied type arguments do not satisfy the constraints of their corresponding type parameters.

3.5 Named Types

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

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

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

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

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

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

the type reference