The apparent type of a type *T* is defined as follows:

- If *T* is the primitive type Number, Boolean, or String, the apparent type of *T* is the augmented form (as defined below) of the global interface type 'Number', 'Boolean', or 'String'.
- if *T* is an enum type, the apparent type of *T* is the augmented form of the global interface type 'Number'.
- If *T* is an object type, the apparent type of *T* is the augmented form of *T*.
- if *T* is a type parameter, the apparent type of *T* is the augmented form of the base constraint (section 3.4.1) of *T*.
- Otherwise, the apparent type of *T* is *T* itself.

The augmented form of an object type *T* adds to *T* those members of the global interface type 'Object' that aren't hidden by members in *T*. Furthermore, if *T* has one or more call or construct signatures, the augmented form of *T* adds to *T* the members of the global interface type 'Function' that aren't hidden by members in *T*. Members in *T* 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, a type's apparent type is a subtype of the 'Object' or 'Function' interface unless the type defines members that are incompatible with those of the 'Object' or 'Function' interface—which, for example, occurs if the type defines a property with the same name as a property in the 'Object' or 'Function' interface but with a type that isn't a subtype of that in the 'Object' or 'Function' interface.

Some examples:

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

The last assignment is an error because the apparent type of the object literal has a 'toString' method that isn't compatible with that of 'Object'.

3.8.2 Type and Member Identity

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

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