BACKGROUND RESEARCH

(reference mainly from the handbook of TS)

**TYPE INFERENCES**

* **BASICS**

Although there is sometimes no explicit type annotation in TypeScript code, type inference is still used to produce the information about the data type.

Consider the following case where there is no explicit type annotation provided.

var x = 10;

Although there is no type annotation provided to directly tell the data type of the variable x, it is clear that the type of the variable x is inferred to be number. This type of basic inference occurs when variables and members are initialized., or return types of functions are determined, or default values are set for parameters.

* **BEST COMMON TYPE**

Suppose a type inference is generated when multiple expressions are provided, the “Best common type” feature of type inference is used.

Consider the below case where multiple data types are used to form an array called.

var car1 = [new Honda(), new Jaguar(), new Car\_company()];

As you can see above, all three elements in the array are different in terms of their data type. The best common type algorithm checks all the types of elements in the array and “picks the type that is compatible with all the other candidates.” (TypeScript Handbook) Suppose the super type of the class Honda and class Jaguar are Car\_company, the algorithm will pick Car\_company as the type of the variable car due to its “compatibility.”

How about the case where no one type is a super type of the others? Consider the case below:

var car2 = [new Honda(), new Jaguar(), new Mazda(), new BMW()];

Suppose the class Car\_company is a super type of all of the elements in the array car2, no element in the array has the type of Car\_company. In order to revise this correctly, the data type should be provided explicitly as following.

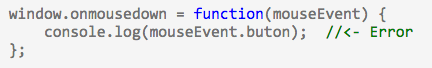
var car2: Car\_company []= [new Honda(), new Jaguar(), new Mazda(), new BMW()];

When no best common type is found by the algorithm, the type inference results in the “empty object type, {}.” Due to the lack of members of this particular type, errors will occur when “attempting to use any properties of it.” This allows users to use the object in a “type-agnostic manner” while using one of the TypeScript’s advantages, Type-safety.

* CONTEXTUAL TYPE

Contextual typing is operated when “the type of an expression is implied by its location. “

Consider the following example below (directly from the handbook):



“For the code above to give the type error, the TypeScript type checker used the type of the Window.onmousedown function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the mouseEvent parameter. If this function expression were not in a contextually typed position, the mouseEvent parameter would have type any, and no error would have been issued.”

If the contextually typed expression includes explicit type information, the contextual type is ignored as shown below:



Contextual typing is applied in various cases including “arguments to function calls, right hand side of the assignments, type assertions, members of object and array literals, and return statements.”

**WRITING .d.ts files**

Declaration files (.d.ts) allows users to “describe the shape of the library” when using an external JavaScript library, or new host API. It allows users to reuse the declaration files across the platform without changing JavaScript files.

**GUIDELINES**

The most ideal way to write a .d.ts file is to begin with writing the library documentation. This allows the completed .d.ts file to be clear to understand by minimizing any implementation details that can lead to misleading.

**NAMESPACING**

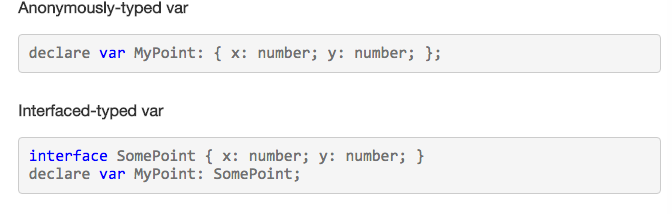
Declaration file writer has an option to put the types inside a module or not when defining interfaces. If a particular type is unlikely to be “referenced directly”, or cannot be named with an ideal name (unique), modules can be used to prevent any possible collision with other types. However, in a case where it is very likely to see consumers declaring variables or parameters of that type often, placing the interface in the global namespace is preferred.

**CALLBACKS (not sure + need more details)**

When writing the function signatures for the case when the library takes a function as a parameter, it is important to avoid marking the parameters as optional. “The right way to think of this is “What parameters will be provided?” instead of “What parameters will be consumed?””

**EXTENSIBLITY AND DECLARATION MERGING**

There are two ways to declare a variable by using an anonymous type or an interface type.



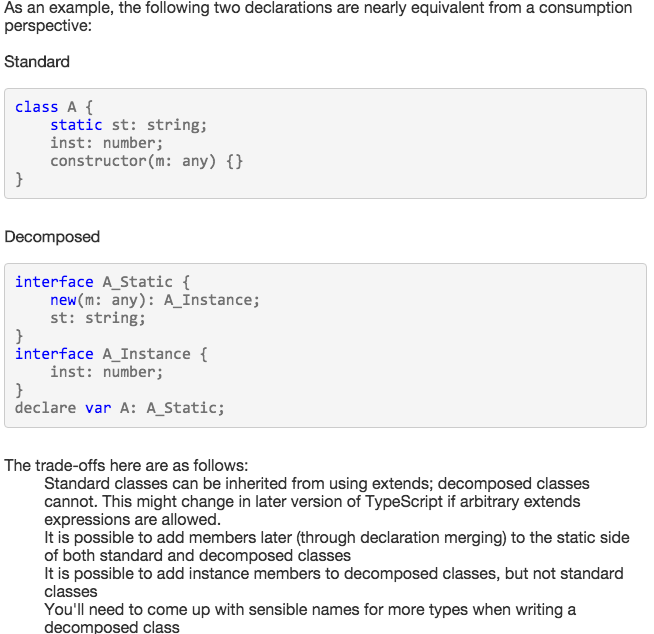
(Directly from the handbook)

By using interface merging, we can easily extend an interface as shown below:



**CLASS DECOMPOSITION**

There are two types created by classes in TypeScript: Instance type and Constructor Function type. The instance type defines the members of the class instance while the constructor function type defines the members of the constructor function. The constructor function type is also referred as the “static side” type due to its inclusion of static members.



**NAMING CONVENTIONS**

In general, interfaces should not be prefixed with I (e.g. IPhone) due to the complexity of the interface concept in TypeScript.

\*NOTE: examples need to be changed.

Details has to be added.

Format has to be revised.