Interfaces

Interfaces introduction

Interfaces in TypeScript are used to *name* the *shape* of `object`s.

Our first shape

```
function printAge(aged: { age: number }){
    console.log(aged.age);
}

const foo = { age: 42, name: 'Foo' };
printAge(foo);
```

The *shape* of `aged` in this example is `{ age: number }`.
Object `foo` complies to this shape

`{ age: number }` is an *anonymous interface*

Structural type system

TypeScript's type system is a *structural type system*.

> In structural typing, an element is considered to be compatible with another if, for each feature within the second element's type, a corresponding and identical feature exists in the first element's type.
> https://en.wikipedia.org/wiki/Structural_type_system

Interface

```
interface Aged {
    age: number;
}

function printAge(aged: Aged) {
    console.log(aged.age);
}

const frank = { age: 23, name: 'Frank' };
printAge(frank);
```

Naming an interface is just creating an *alias* for the shape.

Explicit implementation

```
interface Aged {
    age: number;
}

const frank: Aged = { age: 23, name: 'Frank' };
// => Type '{ age: number; name: string; }' is not assignable to type
'Aged'.
//       Object literal may only specify known properties,
//        and 'name' does not exist in type 'Aged'.
printAge(frank);
```

A type annotation will demand that the shape of an object does not have excess properties.

Optional properties

You can specify optional properties using ?

```
interface Options {
    files?: string[];
    watch?: boolean;
    project?: string;
}

const options: Options = {};
options.watch = true;
options.files = 'error';
// => Type '"error"' is not assignable to type 'string[]'.
options.unknown = 'error';
// => Property 'unknown' does not exist on type 'Options'.
```

Useful when defining config objects, a common JavaScript pattern.

Readonly properties

```
interface FrozenPoint {
    readonly x: number;
    readonly y: number;
}

const origin: FrozenPoint = { x: 0, y: 0 };

origin.x = 3;
```

```
// =>  error: Cannot assign to 'x' because it is a constant or a
// read-only property.
```

This is also implemented for `Object.freeze`:

```
interface Point { x: number; y: number; }
const origin: Point = { x: 0, y: 0 };

const readonlyOrigin = Object.freeze(origin);
readonlyOrigin.x = 34;
// =>  error: Cannot assign to 'x' because it is a constant or a
// read-only property.
```

Index accessors

It is also possible to add an index accessors

```
interface Person {
    firstName: string;
    age: number;
}
const personsByFirstName: {
    [firstName: string]: Person
} = { };

personsByFirstName['Foo'] = { firstName: 'Foo', age: 25 }; // => OK
personsByFirstName.bar = { firstName: 'bar', age: 23 };
```

Function shapes

It is even possible to define the shape of a function using an interface

```
interface BinaryOperation {
    (a: number, b: number): number;
}

const sum: BinaryOperation = (a, b) => a + b;
    // => OK, a and b are inferred as numbers now

const stringified: BinaryOperation = (a, b) => a + '+' + b;
// => Error: Type '(a: number, b: number) => string' is not assignable to
type
//   'BinaryOperation'.
//     Type 'string' is not assignable to type 'number'
```

Extending interfaces

Interfaces can extend each other

```
interface Point {
    x: number;
    y: number;
}

interface ColoredPoint extends Point {
    color: 'red' | 'blue' | 'green';
}
```

Add to an existing interface

You can also add to an existing interface

```
interface Person {
    name: string;
}

interface Person {
    id: number;
}

const han: Person = {
    name: 'Han Solo',
    id: 21
}
```

Question: When can this be useful?

Recap

Interfaces are a powerful tool to describe the rich shapes that are present in the dynamic world of JavaScript