TypeScript
Cheat Sheet

Interface

Key points

Used to describe the shape of objects, and can be extended by others.

Almost everything in JavaScript is an object and **`interface`** is built to match their runtime behavior.

Built-in Type Primitives

```
boolean, string, number,
undefined, null, any,
unknown, never, void,
bigint, symbol
```

Common Built-in JS Objects

```
Date, Error, Array, Map,
Set, Regexp, Promise
```

Type Literals

Object:
`{ field: string }`
Function:
`(arg: number) => string`
Arrays:
`string[] or Array<string>`
Tuple:
`[string, number]`

Avoid

Object, String, Number, Boolean

Common Syntax

```
interface JSONResponse extends Response, HTTPAble {
  version: number;

  /** In bytes */
  payloadSize: number;

  outOfStock?: boolean;

  update: (retryTimes: number) => void;
  update(retryTimes: number): void;

  (): JSONResponse

  new(s: string): JSONResponse;

  [key: string]: number;

  readonly body: string;
}
```

Optionally take properties from existing interface or type

JSDoc comment attached to show in editors

This property might not be on the object

These are two ways to describe a property which is a function

You can call this object via () - (functions in JS are objects which can be called)

You can use new on the object this interface describes

Any property not described already is assumed to exist, and all properties must be numbers

Tells TypeScript that a property can not be changed

Generics

Declare a type which can change in your interface

```
interface APICall<Response> {
  data: Response
}
```

Type parameter

Used here

Usage

```
const api: APICall<ArtworkCall> = ...
api.data // Artwork
```

You can constrain what types are accepted into the generic parameter via the extends keyword.

```
interface APICall<Response extends { status: number }> {
  data: Response
}

const api: APICall<ArtworkCall> = ...
api.data.status
```

Sets a constraint on the type which means only types with a 'status' property can be used

Overloads

A callable interface can have multiple definitions for different sets of parameters

```
interface Expect {
    (matcher: boolean): string
    (matcher: string): boolean;
}
```

Get & Set

Objects can have custom getters or setters

```
interface Ruler {
    get size(): number
    set size(value: number | string);
}
```

Usage

```
const r: Ruler = ...
r.size = 12
r.size = "36"
```

Extension via merging

Interfaces are merged, so multiple declarations will add new fields to the type definition.

```
interface APICall {
  data: Response
}

interface APICall {
  error?: Error
}
```

Class conformance

You can ensure a class conforms to an interface via `implements`:

```
interface Syncable { sync(): void }
class Account implements Syncable { ... }
```