TypeScript
Cheat Sheet

Type

Key points

Full name is "type alias" and are used to provide names to type literals

Supports more rich type-system features than interfaces.

Type vs Interface

- Interfaces can only describe object shapes
- Interfaces can be extended by declaring it multiple times
- In performance critical types interface comparison checks can be faster.

Think of Types Like Variables

Much like how you can create variables with the same name in different scopes, a type has similar semantics.

Build with Utility Types

TypeScript includes a lot of global types which will help you do common tasks in the type system. Check the site for them.

Object Literal Syntax

```ts
type JSONResponse = {
  version: number;                        // Field
  /** In bytes */                         // Attached docs
  payloadSize: number;                    //
  outOfStock?: boolean;                   // Optional
  update: (retryTimes: number) => void;   // Arrow func field
  update(retryTimes: number): void;       // Function
  (): JSONResponse                        // Type is callable
  [key: string]: number;                  // Accepts any index
  new (s: string): JSONResponse;          // Newable
  readonly body: string;                  // Readonly property
}
```

Terser for saving space, see Interface Cheat Sheet for more info, everything but 'static' matches.

Primitive Type

Useful for documentation mainly

```ts
type SanitizedInput = string;
type MissingNo = 404;
```

Object Literal Type

```ts
type Location = {
  x: number;
  y: number;
};
```

Tuple Type

A tuple is a special-cased array with known types at specific indexes.

```ts
type Data = [
    location: Location,
    timestamp: string
];
```

Union Type

Describes a type which is one of many options, for example a list of known strings.

```ts
type Size =
   "small" | "medium" | "large"
```

Intersection Types

A way to merge/extend types

```ts
type Location =
  { x: number } & { y: number }
// { x: number, y: number }
```

Type Indexing

A way to extract and name from a subset of a type.

```ts
type Response = { data: { ... } }

type Data = Response["data"]
// { ... }
```

Type from Value

Re-use the type from an existing JavaScript runtime value via the `typeof` operator.

```ts
const data = { ... }
type Data = typeof data
```

Type from Func Return

Re-use the return value from a function as a type.

```ts
const createFixtures = () => { ... }
type Fixtures =
  ReturnType<typeof createFixtures>

function test(fixture: Fixtures) {}
```

Type from Module

```ts
const data: import("./data").data
```

These features are great for building libraries, describing existing JavaScript code and you may find you rarely reach for them in mostly TypeScript applications.

Mapped Types

Acts like a map statement for the type system, allowing an input type to change the structure of the new type.

```ts
type Artist = { name: string, bio: string }

type Subscriber<Type> = {
  [Property in keyof Type]:
    (newValue: Type[Property]) => void
}
type ArtistSub = Subscriber<Artist>
// { name: (nv: string) => void,
//   bio: (nv: string) => void }
```

Loop through each field in the type generic parameter "Type"

Sets type as a function with original type as param

Conditional Types

Acts as "if statements" inside the type system. Created via generics, and then commonly used to reduce the number of options in a type union.

```ts
type HasFourLegs<Animal> =
   Animal extends { legs: 4 } ? Animal
    : never

type Animals = Bird | Dog | Ant | Wolf;
type FourLegs = HasFourLegs<Animals>
// Dog | Wolf
```

Template Union Types

A template string can be used to combine and manipulate text inside the type system.

```ts
type SupportedLangs =  "en" | "pt" | "zh";
type FooterLocaleIDs = "header" | "footer";

type AllLocaleIDs =
  `${SupportedLangs}_${FooterLocaleIDs}_id`;
// "en_header_id" | "en_footer_id"
 | "pt_header_id" | "pt_footer_id"
 | "zh_header_id" | "zh_footer_id"
```