TypeScript
Cheat Sheet

Control Flow Analysis

Key points

CFA nearly always takes a union and reduces the number of types inside the union based on logic in your code.

Most of the time CFA works inside natural JavaScript boolean logic, but there are ways to define your own functions which affect how TypeScript narrows types.

If Statements

Most narrowing comes from expressions inside if statements, where different type operators narrow inside the new scope

typeof (for primitives)

```
const input = getUserInput()
input // string | number

if (typeof input === “string”) {
  input // string
}
```

"property" in object (for objects)

```
const input = getUserInput()
input // string | { error: ... }

if (“error” in input) {
  input // { error: ... }
}
```

instanceof (for classes)

```
const input = getUserInput()
input // number | number[]

if (input instanceof Array) {
  input // number[]
}
```

type-guard functions (for anything)

```
const input = getUserInput()
input // number | number[]

if (Array.isArray(input)) {
  input // number[]
}
```

Expressions

Narrowing also occurs on the same line as code, when doing boolean operations

```
const input = getUserInput()
input // string | number

const inputLength =
  (typeof input === "string" && input.length) || input
                               // input: string
```

Discriminated Unions

```
type Responses =
  | { status: 200, data: any }
  | { status: 301, to: string }
  | { status: 400, error: Error }
```

All members of the union have the same property name, CFA can discriminate on that.

Usage

```
const response = getResponse()
response // Responses

switch(response.status) {
  case 200: return response.data
  case 301: return redirect(response.to)
  case 400: return response.error
}
```

Type Guards

A function with a return type describing the CFA change for a new scope when it is true.

```
function isErrorResponse(obj: Response): obj is APIErrorResponse {
   return obj instanceof APIErrorResponse
}
```

Return type position describes what the assertion is

Usage

```
const response = getResponse()
response // Response | APIErrorResponse

if (isErrorResponse(response)) {
   response // APIErrorResponse
}
```

Assertion Functions

A function describing CFA changes affecting the current scope, because it throws instead of returning false.

```
function assertResponse(obj: any): asserts obj is SuccessResponse {
  if (!(obj instanceof SuccessResponse)) {
    throw new Error(“Not a success!”)
  }
}
```

Usage

```
const res = getResponse()
res // SuccessResponse | ErrorResponse

assertResponse(res)

res // SuccessResponse
```

Assertion functions change the *current* scope or throw

Assignment

Narrowing types using 'as const'

Subfields in objects are treated as though they can be mutated, and during assignment the type will be 'widened' to a non-literal version. The prefix 'as const' locks all types to their literal versions.

```
const data1 = {             typeof data1 = {
   name: "Zagreus"   »        name: string
}                           }

const data2 = {             typeof data2 = {
   name: "Zagreus"   »        name: "Zagreus"
} as const                  }
```

Tracks through related variables

```
const response = getResponse()
const isSuccessResponse
    = res instanceof SuccessResponse

if (isSuccessResponse)
  res.data // SuccessResponse
```

Re-assignment updates types

```
let data: string | number = ...
data // string | number
data = "Hello"
data // string
```

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 { ... }
```

TypeScript Cheat Sheet Class

Key points

A TypeScript class has a few type-specific extensions to ES2015 JavaScript classes, and one or two runtime additions.

Creating an class instance

```
class ABC { ... }
const abc = new ABC()
```

Parameters to the new ABC come from the constructor function.

private x vs #private

The prefix private is a type-only addition, and has no effect at runtime. Code outside of the class can reach into the item in the following case:

```
class Bag {
  private item: any
}
```

Vs #private which is runtime private and has enforcement inside the JavaScript engine that it is only accessible inside the class:

```
class Bag { #item: any }
```

'this' in classes

The value of 'this' inside a function *depends on how the function is called*. It is not guaranteed to always be the class instance which you may be used to in other languages.

You can use 'this parameters', use the bind function, or arrow functions to work around the issue when it occurs.

Type and Value

Surprise, a class can be used as both a type or a value.

```
const a:Bag = new Bag()
```

Type

Value

So, be careful to not do this:

```
class C implements Bag {}
```

Common Syntax

```
class User extends Account implements Updatable, Serializable {
  id: string;                      // A field
  displayName?: boolean;           // An optional field
  name!: string;                   // A 'trust me, it's there' field
  #attributes: Map<any, any>;      // A private field
  roles = ["user"];                // A field with a default
  readonly createdAt = new Date() // A readonly field with a default

  constructor(id: string, email: string) {
    super(id);
    this.email = email;
    ...
  };

  setName(name: string) { this.name = name }
  verifyName = (name: string) => { ... }

  sync(): Promise<{ ... }>
  sync(cb: ((result: string) => void)): void
  sync(cb?: ((result: string) => void)): void | Promise<{ ... }> { ... }

  get accountID() { }
  set accountID(value: string) { }

  private makeRequest() { ... }
  protected handleRequest() { ... }

  static #userCount = 0;
  static registerUser(user: User) { ... }

  static { this.#userCount = -1 }
}
```

Subclasses this class

Ensures that the class conforms to a set of interfaces or types

The code called on 'new'

In strict: true this code is checked against the fields to ensure it is set up correctly

Ways to describe class methods (and arrow function fields)

A function with 2 overload definitions

Getters and setters

Private access is just to this class, protected allows to subclasses. Only used for type checking, public is the default.

Static fields / methods

Static blocks for setting up static vars. 'this' refers to the static class

Generics

Declare a type which can change in your class methods.

```
class Box<Type> {
  contents: Type
  constructor(value: Type) {
    this.contents = value;
  }
}
const stringBox = new Box("a package")
```

Class type parameter

Used here

These features are TypeScript specific language extensions which may never make it to JavaScript with the current syntax.

Parameter Properties

A TypeScript specific extension to classes which automatically set an instance field to the input parameter.

```
class Location {
  constructor(public x: number, public y: number) {}
}
const loc = new Location(20, 40);
loc.x // 20
loc.y // 40
```

Abstract Classes

A class can be declared as not implementable, but as existing to be subclassed in the type system. As can members of the class.

```
abstract class Animal {
  abstract getName(): string;
  printName() {
    console.log("Hello, " + this.getName());
  }
}
class Dog extends Animal { getName(): { ... } }
```

Decorators and Attributes

You can use decorators on classes, class methods, accessors, property and parameters to methods.

```
import {
  Syncable, triggersSync, preferCache, required
} from "mylib"

@Syncable
class User {
  @triggersSync()
  save() { ... }

  @preferCache(false)
  get displayName() { ... }

  update(@required info: Partial<User>) { ... }
}
```

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 mutliple 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

```
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

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

Object Literal Type

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

Tuple Type

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

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

Union Type

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

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

Intersection Types

A way to merge/extend types

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

Type Indexing

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

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

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

Type from Value

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

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

Type from Func Return

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

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

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

Type from Module

```
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.

```
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.

```
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.

```
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"
```