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

Subclasses this class

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

```
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 }
}
```

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>) { ... }
}
```