Cheatsheets / **Learn TypeScript**

Advanced Object Types

TypeScript Interface Type

TypeScript allows you to specifically type an object using an interface that can be reused by multiple objects. To create an interface, use the `interface` keyword followed by the interface name and the typed object.

```
interface Publication {
  isbn: string;
  author: string;
  publisher: string;
}
```

TypeScript Interface Define Objects Only

In TypeScript, type aliases can define composite types such as objects and unions as well as primitive types such as numbers and strings; interface, however, can only define objects. Interface is useful in typing objects written for object-oriented programs.

```
// Type alias can define a union type
type ISBN = number | string;

// Type alias can define an object type
type PublicationT = {
  isbn: ISBN;
  author: string;
  publisher: string;
}

// Interface can only define an object
type
interface PublicationI {
  isbn: ISBN;
  author: string;
  publisher: string;
}
```

TypeScript Interface for Classes

To apply a TypeScript interface to a class, add the `implements` keyword after the class name followed by the interface name. TypeScript will check and ensure that the object actually

```
interface Shape {
  area: number;
  computeArea: () => number;
```

implements all the properties and methods defined inside the interface.

```
}

const PI: number = 22/7 ;

// Circle class implements the Shape
interface
class Circle implements Shape {
  radius: number;
  area: number;
  constructor(radius: number) {
    this.radius = radius;
    this.area = this.computeArea();
  }
  computeArea = (): number => {
    return PI * this.radius *
this.radius;
  }
}

let target = new Circle(3);
console.log(target.area.toFixed(2));
// Prints "28.29"
```

TypeScript Nested Interface

TypeScript allows both type aliases and interface to be nested. An object typed with a nested interface should have all its properties structured the same way as the interface definition.

```
// This is a nested interface
interface Course {
  description: {
    name: string;
    instructor: {
      name: string;
    }
    prerequisites: {
      courses: string[];
    }
  }
}
```

```
class myCourse implements Course {
  description = {
    name: '',
    instructor: {
      name: ''
    },
    prerequisites: {
      courses: []
    }
  }
}
```

TypeScript Nesting Interfaces Inside an Interface

Since interfaces are composable, TypeScript allows you to nest interfaces within an interface.

```
// Date is composed of primitive types
interface Date {
  month: number;
  day: number;
  year: number
}

// Passport is composed of primitive
types and nested with another interface
interface Passport {
  id: string;
  name: string;
  citizenship: string;
  expiration: Date;
}

// Ticket is composed of primitive
types and nested with another interface
interface Ticket {
  seat: string;
  expiration: Date;
}

// TravelDocument is nested with two
```

```
other interfaces
interface TravelDocument {
  passport: Passport;
  ticket: Ticket;
}
```

TypeScript Interface Inheritance

Like JavaScript classes, an interface can inherit properties and methods from another interface using the `extends` keyword. Members from the inherited interface are accessible in the new interface.

```
interface Brand {
  brand: string;
}

// Model inherits property from Brand
interface Model extends Brand {
  model: string;
}

// Car has a Model interface
class Car implements Model {
  brand;
  model;
  constructor(brand: string, model:
string) {
    this.brand = brand;
    this.model = model;
  }
  log() {
    console.log(`Drive a ${this.brand}
${this.model} today!`);
  }
}

const myCar: Car = new Car('Nissan',
'Sentra');
myCar.log(); // Prints "Drive a Nissan
Sentra today!"
```

TypeScript Interface Index Signature

Property names of an object are assumed to be strings, but they can also be numbers. If you don't know in advance the types of these property names, TypeScript allows you to use an index signature to specify the type for the property name inside an object. To specify an index signature, use square brackets, `[...]`, to surround the type notation for the property name.

```
interface Code {
  [code: number]: string;
}
const codeToStates: Code = {603: 'NH',
617: 'MA'};

interface ReverseCode {
  [code: string]: number;
}
const stateToCodes: ReverseCode =
{'NH': 603, 'MA': 617};
```

TypeScript Interface Optional Properties

TypeScript allows you to specify optional properties inside an interface. This is useful in situations where not all object properties have values assigned to them. To indicate if a property is optional, append a `?` symbol after the property name before the colon, `:`.

```
interface Profile {
  name: string;
  age: number;
  hobbies?: string[];
}

// The property, hobbies, is optional,
but name and age are required.
const teacher: Profile = {name: 'Tom
Sawyer', age: 18};
```