TypeScript Generics

Generics allow creating 'type variables' which can be used to create classes, functions & type aliases that don't need to explicitly define the types that they use. Generics makes it easier to write reusable code.

Functions:

Generics with functions help make more generalized methods which more accurately represent the types used and returned.

```
function createPair<S, T>(v1: S, v2: T): [S, T]{

 return [v1, v2];  }

console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

Classes:

Generics can be used to create generalized classes, like Map.

```
class NamedValue<T> {
  private _value: T | undefined;
  constructor(private name: string) {}
  public setValue(value: T) {
    this._value = value;
  }
  public getValue(): T | undefined {
    return this._value;
  }
  public toString(): string {
    return `${this.name}: ${this._value}`;
  }
}
const value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

Interface:

Generics can be used to create generalized Interface too:

```
interface KeyPair<T, U> {
    key: T;
    value: U;
}
let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK
let kv2: KeyPair<number, number> = { key:1, value:12345 }; // OK
```

Type Aliases

Generics in type aliases allow creating types that are more reusable.

```
type Wrapped<T> = { value: T };
const wrappedValue: Wrapped<number> = { value: 10 };
```

Extends

Constraints can be added to generics to limit what's allowed. The constraints make it possible to rely on a more specific type when using the generic type.

```
function createLoggedPair<S extends string | number, T extends string | number>(v1: S, v2: T): [S, T] {
  console.log(`creating pair: v1='${v1}', v2='${v2}'`);
  return [v1, v2];
}
```

References:

https://www.w3schools.com/typescript/typescript_basic_generics.php

https://www.digitalocean.com/community/tutorials/how-to-use-generics-in-typescript