TypeScript Basic Generics

TypeScript **Generic Types** can be used by programmers when they need to create reusable components because they are used to create components that work with various data types and this provides type safety. The reusable components can be classes, functions, and interfaces. TypeScript generics can be used in several ways like function, class, and interface 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.

Syntax:

```
function functionName<T> (returnValue : T) : T {
    return returnValue;
}
```

Functions

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

Example

```
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}
console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

TypeScript can also infer the type of the generic parameter from the function parameters.

Classes

Generics can be used to create generalized classes, like Map

Example

```
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}`;
  }
}

let value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

TypeScript can also infer the type of the generic parameter if it's used in a constructor parameter.

Default Value

Generics can be assigned default values which apply if no other value is specified or inferred.

Example

```
class NamedValue<T = string> {
  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}`;
  }
}

let value = new NamedValue('myNumber');
value.setValue('myValue');
console.log(value.toString()); // myNumber: myValue
```

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.

Example

```
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];
}
```

This can be combined with a default value.

Below are two more examples of basic generic type usage

Example 1: In this example, we have created a generic function that can accept any type of data, we need to pass the value in the parameter by giving them any kind of data type and then we can reverse that value by the use of the reverseArray function.

```
function reverseArray<T>(array: T[]): T[] {
    return array.reverse();
}
const strArray: string[] = reverseArray(["Java", "Python", "C++"]);
const numArray: number[] = reverseArray([1, 2, 3, 4, 5]);
const boolArray: boolean[] = reverseArray([false, true]);
console.log(strArray);
console.log(numArray);
console.log(boolArray);
```

Output:

```
[ 'C++', 'Python', 'Java' ]
[ 5, 4, 3, 2, 1 ]
[ true, false ]
```

Example 2: In this example, we have created a generic interface by using that we are creating object and string type of value and printing them in the console.

```
interface Resource<T> {
    id: number;
    resourceName: string;
    data: T;
}
const person: Resource<object> = {
    // Generic interface with objects
    id: 1,
    resourceName: 'Person',
    data: {
        name: 'John',
        age: 25,
```

```
    }
}

console.log(person);
const employee: Resource<string[]> =
{
    // Generic interface with strings array
    id: 2,
    resourceName: 'Employee',
    data: ['Employee 1', 'Employee 1']
}
console.log(employee);
```

Output:


```
 {
  "id": 1,
  "resourceName": "Person",
  "data": {
    "name": "John",
    "age": 25
  }
}
{
  "id": 2,
  "resourceName": "Employee",
  "data": [
    "Employee 1",
    "Employee 1"
  ]
}
```