**TYPESCRIPT GENERICS**

Generics in typescript is used to write reusable code components that can work with any datatype without being restricted to a particular datatype. The datatypes of generics is specified by the user while using them.

* Generics in typescript is the type that is connected to another type.
* Generics always starts with capital letter.
* Generics is used to provide information about the incoming data in typescript.
* Array<datatype>, Promise<datatype>, Partial<datatype>, Readonly<datatype> are some of the built in generics types in typescript.
* //? Generics.
* //\* Generics always starts with Capital letter.
* //\* Generics are typescript specific feature not vailable in javascript.
* //\* Generics in typescript is a type that is connected to another datatype.
* //\* Generics priovide typescript additional information about the incoming data and the operations on them.
* //\* Array, Promise, Partial, Readonly are some of the generic types available in typescript.
* //\* Array<> generic type
* // Gives as information about the type of data to be stored in an array.
* const users: Array<string> = ['max', 'micky']; //? The above line converted to generic notation.
* //\* Promsie<> generic type
* // Gives un infomation about the type of data we are going to receive as promise completed.
* const promise: Promise<string> = new Promise((resolve, reject) => {
* setTimeout(() => {
* if (true) {
* resolve('result done.')
* } else {
* reject('result pending');
* }
* }, 2000)
* })
* promise.then((data) => {
* data.split(' ')
* })
* //\* Partial<> generic type
* // This generic type is used to make all the mandatory porperties of an object optional temporarily.
* interface ExamResult { maths: number, english: number, science: number } //? all the properties are mandatory.
* function makeExamResult(mathsScore: number, englishScore: number, scienceScore: number): ExamResult {
* let result: Partial<ExamResult> = {};
* result.maths = mathsScore;
* result.english = englishScore;
* result.science = scienceScore;
* return <ExamResult>result //? Type casting.
* }
* //\* Readonly<> generic
* // This generic type is used to to make a variable readonly.
* const readonlyArray: Readonly<string[]> = ['Harshit', 'Bhawsar']
* //! readonlyArray.push('hahaa') NOT ALLOWED
* //! readonlyArray.pop() NOT ALLOWED

# **Creating Generic functions:**

* Generic functions are made to let typescript know the incoming data from the function.
* In Generic functions the variables defined inside the <> works as the placeholder datatype for the arguments.
* //? Generic fucntion
* //\* we write generic functions when we are not sure about the structure of data we are going to receive.
* // Here T and U works as the Placeholder for whatever type of arguments going to receive.
* function combineObject<T, U>(obj1: T, obj2: U) {
* return { ...obj1, ...obj2 }
* }
* const result = combineObject({ name: "Harshit" }, { age: 24 });
* console.log(`The generic function result => `, result.name, result.age, result)

# **Constraints In Generic functions:**

* Constraints in generic functions are defined by using the **extends** keyword.
* Constraints in generic functions are used to tell typescript that which datatype is accepted irrespective of its structure.
* //? Constraints in generic function.
* //\* we can use the extends keyword to tell typescript that the generic type should accepts a particular type.
* //\* Without contraints (can accept any type as argument, but silenty fails.)
* function combineObject<T, U>(obj1: T, obj2: U) {
* return { ...obj1, ...obj2 }
* }
* console.log('The generic without constarints => ', combineObject({ name: 'Harshit' }, 30))  // output {name: 'Harshit'}
* //\* With Constraints (argumnets should be an object not matter what structure they have)
* function mergeObject<T extends object, U extends object>(obj1: T, obj2: U) {
* return { ...obj1, ...obj2 }
* }
* console.log('The generic with constarints => ', mergeObject({ name: 'Harshit' }, { age: 24 }))  // output {name: 'Harshit'}
* The **keyof**  constraint in generic is used tom define that the given value must be the key of the object.
* /\* The "keyof" constaint in generic
* // It defines that the passed value must be the key of the object.
* function extractTheValue<T extends object, U extends keyof T>(obj: T, key: U) {
* return obj[key];
* }
* //! extractTheValue({}, 'name') Error (key is not present in object.)
* extractTheValue({ name: 'Harshit' }, 'name')

# **Creating Generic Classes:**

* Generic classes can be created in the same way we create generic functions.
* The generic types <T, U> acts as a placeholder for the type the user define at the time of using the class/function.
* //? Generic classes.
* //\* our example
* class PriceOfElement<T> {
* constructor(public name: string, public price: T) { }
* }
* const car = new PriceOfElement('Prius', 5000)
* const bike = new PriceOfElement('Hayabusa', '18 lakhs');
* //\* Lecturer example
* class DataStorage<T extends string | number | boolean> {
* private data: T[]
* constructor() {
* this.data = []
* }
* addData(element: T) { this.data.push(element) };
* removeData(element: T) { this.data = this.data.filter(el => el !== element) };
* printData() { console.log(this.data) }
* }
* const textStorage = new DataStorage<string>();
* textStorage.addData('Harshit');
* textStorage.addData('Kumar');
* textStorage.addData('Bhawsar');
* textStorage.removeData('Kumar');
* textStorage.printData();
* const numericStorage = new DataStorage<number>();
* numericStorage.addData(50);
* numericStorage.addData(34);
* numericStorage.addData(10);
* numericStorage.removeData(50);
* numericStorage.printData();
* const alphaNumericStorage = new DataStorage<number | string>();
* alphaNumericStorage.addData(50);
* alphaNumericStorage.addData('Harshit');
* alphaNumericStorage.addData(10);
* alphaNumericStorage.removeData(50);
* alphaNumericStorage.printData();

# **Creating Generic Interfaces:**

//? Generic interfaces

interface Count<T> {

    count: T

}

const numericCount: Count<number> = { count: 23 }

const textCount: Count<string> = { count: 'Twele' }

# **Generics Vs Unions:**

* Generics are used when we have to lock in to a particular type, while unions are used when we have to work with different types of data  [simultaneously](https://www.bing.com/search?q=simultaneously&FORM=AWRE).