

TypeScript Utility Types এবং Generic Types - Complete Guide

Table of Contents

1. Utility Types

- Partial
- Required
- Pick
- Omit
- Record

2. Generic Types

- Generic কি এবং কেন প্রয়োজন
 - Generic Functions
 - Generic Interfaces
 - Generic Classes
 - Advanced Generic Concepts
-

Utility Types

TypeScript এর Utility Types হলো built-in type transformations যা existing types থেকে new types তৈরি করতে সাহায্য করে। এগুলো code reusability বাড়ায় এবং type safety নিশ্চিত করে।

1. **Partial<T>**

কি: একটি type এর সব properties কে optional করে দেয়।

কিভাবে ব্যবহার করে:

```
typescript
```

```

interface User {
  id: number;
  name: string;
  email: string;
  age: number;
}

// Partial ব্যবহার করে সব properties optional করা
type PartialUser = Partial<User>;
// Result: { id?: number; name?: string; email?: string; age?: number; }

// Example usage
function updateUser(user: User, updates: Partial<User>): User {
  return { ...user, ...updates };
}

const user: User = { id: 1, name: "John", email: "john@example.com", age: 25 };
const updatedUser = updateUser(user, { name: "John Doe" }); // শুধু name update করা হলো

```

কখন ব্যবহার করে:

- Update operations এ যখন সব fields প্রয়োজন নেই
- Optional configuration objects তৈরি করতে
- Form validation এ partial data handle করতে

2. Required<T>

কি: একটি type এর সব optional properties কে required করে দেয়।

কিভাবে ব্যবহার করে:

typescript

```

interface UserSettings {
  theme?: string;
  notifications?: boolean;
  language?: string;
}

// Required ব্যবহার করে সব properties required করা
type RequiredUserSettings = Required<UserSettings>;
// Result: { theme: string; notifications: boolean; language: string; }

// Example usage
function validateCompleteSettings(settings: Required<UserSettings>) {
  // এখানে guaranteed যে সব properties আছে
  console.log(`Theme: ${settings.theme}`);
  console.log(`Notifications: ${settings.notifications}`);
  console.log(`Language: ${settings.language}`);
}

const completeSettings: Required<UserSettings> = {
  theme: "dark",
  notifications: true,
  language: "bn"
};

```

কখন ব্যবহার করে:

- Configuration validation এ
- Database save operations এ যখন সব fields required
- Form submission এ complete data ensure করতে

3. Pick<T, K>

কি: একটি type থেকে নির্দিষ্ট properties select করে নতুন type তৈরি করে।

কিভাবে ব্যবহার করে:

typescript

```

interface Employee {
  id: number;
  name: string;
  email: string;
  salary: number;
  department: string;
  joinDate: Date;
}

// শুধু id, name, email pick করা
type PublicEmployee = Pick<Employee, 'id' | 'name' | 'email'>;
// Result: { id: number; name: string; email: string; }

// Example usage
function getPublicEmployeeInfo(employee: Employee): PublicEmployee {
  return {
    id: employee.id,
    name: employee.name,
    email: employee.email
  };
}

// API response এ sensitive data hide করতে
const publicInfo: PublicEmployee = {
  id: 1,
  name: "আহমেদ",
  email: "ahmed@company.com"
};

```

কখন ব্যবহার করে:

- API responses এ sensitive data filter করতে
- Component props এ নির্দিষ্ট properties pass করতে
- Database queries এ specific fields select করতে

4. Omit<T, K>

কি: একটি type থেকে নির্দিষ্ট properties বাদ দিয়ে নতুন type তৈরি করে।

কিভাবে ব্যবহার করে:

typescript

```

interface Product {
  id: number;
  name: string;
  price: number;
  createdAt: Date;
  updatedAt: Date;
}

// id, createdAt, updatedAt বাদ দিয়ে
type CreateProductInput = Omit<Product, 'id' | 'createdAt' | 'updatedAt'>;
// Result: { name: string; price: number; }

// Example usage
function createProduct(input: CreateProductInput): Product {
  return {
    id: Date.now(), // auto-generated
    ...input,
    createdAt: new Date(),
    updatedAt: new Date()
  };
}

const newProduct = createProduct({
  name: "ল্যাপটপ",
  price: 50000
});

```

কখন ব্যবহার করে:

- Create operations এ auto-generated fields বাদ দিতে
- Form inputs তৈরি করতে
- API request types তৈরি করতে

5. Record<K, T>

কি: Key-value pair এর জন্য একটি type তৈরি করে যেখানে সব keys একই type এর এবং সব values একই type এর।

কিভাবে ব্যবহার করে:

```
typescript
```

```
// String keys এবং number values
type Scores = Record<string, number>;
const gameScores: Scores = {
  "আলী": 95,
  "ফাতিমা": 87,
  "করিম": 92
};

// Union type keys এবং specific value type
type Status = 'pending' | 'approved' | 'rejected';
type StatusConfig = Record<Status, { color: string; message: string }>;

const statusConfig: StatusConfig = {
  pending: { color: 'yellow', message: 'অপেক্ষমান' },
  approved: { color: 'green', message: 'অনুমোদিত' },
  rejected: { color: 'red', message: 'প্রত্যাখ্যাত' }
};

// Example usage
function getStatusInfo(status: Status) {
  return statusConfig[status];
}
```

কখন ব্যবহার করে:

- Configuration objects তৈরি করতে
- Lookup tables তৈরি করতে
- Dynamic key-value pairs handle করতে
- Enum-like structures তৈরি করতে

Generic Types

Generic কি এবং কেন প্রয়োজন?

Generic Types হলো TypeScript এর একটি powerful feature যা আমাদের flexible এবং reusable code লিখতে সাহায্য করে। এটি type parameters ব্যবহার করে একই code বিভিন্ন types এর সাথে কাজ করতে পারে।

সমস্যা (Generic ছাড়া):

```
typescript
```

```
// Different functions for different types
function getFirstString(arr: string[]): string | undefined {
    return arr[0];
}

function getFirstNumber(arr: number[]): number | undefined {
    return arr[0];
}

function getFirstBoolean(arr: boolean[]): boolean | undefined {
    return arr[0];
}
```

সমাধান (Generic দিয়ে):

```
typescript

function getFirst<T>(arr: T[]): T | undefined {
    return arr[0];
}

// ব্যবহার
const firstString = getFirst(["আপেল", "কলা"]); // Type: string | undefined
const firstNumber = getFirst([1, 2, 3]); // Type: number | undefined
const firstBoolean = getFirst([true, false]); // Type: boolean | undefined
```

Generic Functions

Basic Generic Function:

```
typescript
```

```
function identity<T>(arg: T): T {
  return arg;
}

// Usage
const stringResult = identity("হ্যালো"); // Type: string
const numberResult = identity(42); // Type: number

// Multiple type parameters
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}




const result = pair("বয়স", 25); // Type: [string, number]
```

Generic Function with Constraints:

```
typescript

interface HasLength {
  length: number;
}

function logLength<T extends HasLength>(arg: T): T {
  console.log(`Length: ${arg.length}`);
  return arg;
}

logLength("Hello"); //  Works
logLength([1, 2, 3]); //  Works
logLength({ length: 10, value: "test" }); //  Works
// logLength(123); //  Error: number doesn't have length property
```

Generic Interfaces

```
typescript
```



```
// Generic interface
interface Container<T> {
  value: T;
  getValue(): T;
  setValue(value: T): void;
}

// Implementation
class Box<T> implements Container<T> {
  constructor(private _value: T) {}

  getValue(): T {
    return this._value;
  }

  setValue(value: T): void {
    this._value = value;
  }

  get value(): T {
    return this._value;
  }
}

// Usage
const stringBox = new Box<string>("আমার মান");
const numberBox = new Box<number>(100);

console.log(stringBox.getValue()); // "আমার মান"
console.log(numberBox.getValue()); // 100
```

Generic Interface with Multiple Parameters:

```
typescript
```

```

interface KeyValuePair<K, V> {
  key: K;
  value: V;
}

interface Dictionary<K, V> {
  items: KeyValuePair<K, V>[];
  get(key: K): V | undefined;
  set(key: K, value: V): void;
}

class SimpleDictionary<K, V> implements Dictionary<K, V> {
  items: KeyValuePair<K, V>[] = [];

  get(key: K): V | undefined {
    const found = this.items.find(item => item.key === key);
    return found?.value;
  }

  set(key: K, value: V): void {
    const existingIndex = this.items.findIndex(item => item.key === key);
    if (existingIndex >= 0) {
      this.items[existingIndex].value = value;
    } else {
      this.items.push({ key, value });
    }
  }
}

// Usage
const userScores = new SimpleDictionary<string, number>();
userScores.set("আলী", 95);
userScores.set("সারা", 87);
console.log(userScores.get("আলী")); // 95

```

Generic Classes

typescript

```

class DataStore<T> {
  private data: T[] = [];

  add(item: T): void {
    this.data.push(item);
  }

  get(index: number): T | undefined {
    return this.data[index];
  }

  getAll(): T[] {
    return [...this.data];
  }

  findBy<K extends keyof T>(key: K, value: T[K]): T | undefined {
    return this.data.find(item => item[key] === value);
  }
}

// Usage
interface User {
  id: number;
  name: string;
  email: string;
}

const userStore = new DataStore<User>();
userStore.add({ id: 1, name: "রহিম", email: "rahim@example.com" });
userStore.add({ id: 2, name: "করিম", email: "karim@example.com" });

const user = userStore.findBy("name", "রহিম");
console.log(user); // { id: 1, name: "রহিম", email: "rahim@example.com" }

```

Advanced Generic Concepts

Conditional Types:

typescript

```
type IsArray<T> = T extends any[] ? true : false;
```

```
type Test1 = IsArray<string[]>; // true
```

```
type Test2 = IsArray<number>; // false
```

```
// Practical example
```

```
type ApiResponse<T> = T extends any[]
```

```
  ? { data: T; count: number }
```

```
  : { data: T };
```

```
type UserListResponse = ApiResponse<User[]>;
```

```
// { data: User[]; count: number }
```

```
type SingleUserResponse = ApiResponse<User>;
```

```
// { data: User }
```

Mapped Types with Generics:

```
typescript
```

```
type Optional<T, K extends keyof T> = Omit<T, K> & Partial<Pick<T, K>>;
```

```
interface User {
```

```
  id: number;
```

```
  name: string;
```

```
  email: string;
```

```
  age: number;
```

```
}
```

```
// email এবং age optional করা
```

```
type UserWithOptionalContact = Optional<User, 'email' | 'age'>;
```

```
// Result: { id: number; name: string; email?: string; age?: number; }
```

Generic Utility Functions:

```
typescript
```

```
// Array utilities
```

```
function chunk<T>(array: T[], size: number): T[][] {  
  const chunks: T[][] = [];  
  for (let i = 0; i < array.length; i += size) {  
    chunks.push(array.slice(i, i + size));  
  }  
  return chunks;  
}
```

```
function unique<T>(array: T[]): T[] {  
  return Array.from(new Set(array));  
}
```

```
// Usage
```

```
const numbers = [1, 2, 3, 4, 5, 6];  
const chunked = chunk(numbers, 2); // [[1, 2], [3, 4], [5, 6]]  
  
const duplicates = ["আপেল", "কলা", "আপেল", "আম"];  
const uniqueFruits = unique(duplicates); // ["আপেল", "কলা", "আম"]
```

কখন কোনটা ব্যবহার করবেন?

Utility Types ব্যবহার করুন যখন:

- **Partial:** Update operations, optional configurations
- **Required:** Validation, complete data requirements
- **Pick:** API responses, specific property selection
- **Omit:** Create operations, excluding auto-generated fields
- **Record:** Configuration objects, lookup tables

Generic Types ব্যবহার করুন যখন:

- একই logic বিভিন্ন types এর সাথে কাজ করতে হবে
- Type-safe collections তৈরি করতে হবে
- Reusable utility functions লিখতে হবে
- API response types তৈরি করতে হবে
- Library বা framework তৈরি করতে হবে

Best Practices

1. **Naming Convention:** Generic type parameters এর জন্য `T`, `U`, `V` অথবা descriptive names ব্যবহার করুন
2. **Constraints:** প্রয়োজনে generic constraints ব্যবহার করুন
3. **Default Types:** Generic parameters এর জন্য default values দিন যেখানে সম্ভব
4. **Documentation:** Complex generic types এর জন্য proper documentation লিখুন

এই guide অনুসরণ করে আপনি TypeScript এর Utility Types এবং Generic Types effectively ব্যবহার করতে পারবেন এবং অন্যদের শেখাতে পারবেন।