

i know other type script just don't know the concepts of generics, explain everything in it. give me full in depth course about generics in typescript like generics, in variables like array, object, Boolean etc in classes, functions generics with interfaces or types. everything should be envolved

Comprehensive Guide to TypeScript Generics

Generics enable reusable, type-safe code by allowing you to write components that work with **any type** while preserving type information. Think of them as "type parameters."

1. Generics in Variables

Arrays

typescript

Copy

Download

```
// Generic array types
const numbers: Array<number> = [1, 2, 3]; // Array<number>
const strings: string[] = ["a", "b"]; // Equivalent shorthand

// Mixed types with tuples
const mixed: [string, number] = ["hello", 42]; // Fixed-type array
```

Objects

typescript

Copy

Download

```
// Generic object structure
type KeyValue<T, U> = { key: T; value: U };
const entry: KeyValue<string, number> = { key: "age", value: 30 };
```

Booleans & Primitives

typescript

Copy

Download

```
type Box<T> = { value: T };
const booleanBox: Box<boolean> = { value: true };
const numberBox: Box<number> = { value: 42 };
```

2. Generics in Functions

Basic Function

typescript

Copy

Download

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

const result = identity<string>("Hello"); // Type: string
const inferred = identity(42); // Type inferred as number
```

Multiple Type Parameters

typescript

Copy

Download

```
function merge<T, U>(obj1: T, obj2: U): T & U {
  return { ...obj1, ...obj2 };
}

const merged = merge({ name: "Alice" }, { age: 30 }); // { name: string; age: number }
```

Generic Constraints (Limiting Types)

typescript

Copy

Download

```
interface Lengthy {
  length: number;
}

function logLength<T extends Lengthy>(arg: T): void {
  console.log(arg.length);
}

logLength("hello"); // 5 (string has .length)
logLength([1, 2, 3]); // 3 (array has .length)
// logLength(42); // Error: number has no .length
```

3. Generics in Classes

typescript

Copy

Download

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

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

  dequeue(): T | undefined {
    return this.data.shift();
  }
}

// Usage
const numberQueue = new Queue<number>();
numberQueue.enqueue(1);
numberQueue.enqueue(2);
console.log(numberQueue.dequeue()); // 1 (type: number)
```

```
const stringQueue = new Queue<string>();
stringQueue.enqueue("a");
```

4. Generics with Interfaces & Types

Generic Interfaces

typescript

Copy

Download

```
interface ApiResponse<T> {
  status: number;
  data: T;
}

const userResponse: ApiResponse<{ name: string }> = {
  status: 200,
  data: { name: "Bob" },
};
```

Generic Types (Type Aliases)

typescript

Copy

Download

```
type Pair<T, U> = [T, U];
const coordinates: Pair<number, number> = [10, 20];
```

Extending Interfaces

typescript

Copy

Download

```
interface User {
  id: string;
}

interface Admin<T> extends User {
  permissions: T;
}

const admin: Admin<string[]> = {
  id: "1",
  permissions: ["read", "write"],
};
```

5. Advanced Patterns

Default Generic Types

typescript

Copy

Download

```
function createArray<T = string>(length: number): T[] {
  return Array(length).fill(null);
}

const stringArray = createArray(3); // string[] (default)
const numberArray = createArray<number>(3); // number[]
```

Keyof Constraints

typescript

Copy

Download

```
function getValue<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const person = { name: "Alice", age: 30 };
const age = getValue(person, "age"); // type: number
// getValue(person, "email"); // Error: "email" not in keys
```

Mapped Types

typescript

Copy

Download

```
type Optional<T> = {
  [K in keyof T]?: T[K];
};

type User = { name: string; age: number };
type PartialUser = Optional<User>; // { name?: string; age?: number }
```

6. Real-World Examples

API Wrapper

typescript

Copy

Download

```
async function fetchData<T>(url: string): Promise<T> {
  const response = await fetch(url);
  return response.json();
}

// Usage
interface User { name: string; }
const user = await fetchData<User>("/api/user");
```

State Management

typescript

Copy

Download

```
-----
```

```
class Store<T> {
  private state: T;

  constructor(initialState: T) {
    this.state = initialState;
  }

  getState(): T {
    return this.state;
  }

  update(partial: Partial<T>) {
    this.state = { ...this.state, ...partial };
  }
}

const userStore = new Store({ name: "", age: 0 });
userStore.update({ name: "Charlie" });
```

Key Takeaways

1. **Reusability**: Write functions/classes that work with multiple types.
2. **Type Safety**: Compiler enforces correct usage of generic types.
3. **Constraints**: Use `extends` to restrict generic types (e.g., `T extends SomeInterface`).
4. **Inference**: TypeScript often infers generics automatically.
5. **Utility Types**: Built-ins like `Partial<T>`, `Pick<T, K>`, `Record<K, T>` leverage generics.

Generics turn TypeScript into a powerfully flexible tool while keeping your code type-safe. Practice by converting existing JavaScript utilities to use generics!

function merge<x, y>(obj1: x, obj2: y): x & y { return { ...obj1, ...obj2 }; } const merged = merge({ name: "Alice" }, { age: 30 }); i didn't get this example and if i do function merge<x, y>(obj1: x, obj2: y): x & y { return { obj1, obj2 }; } why this give error

Let

function merge<x, y>(obj1: x, obj2: y): x & y { return { ...obj1, ...obj2 }; } const merged = merge({ name: "Alice" }, { age: 30 }); i didn't get this example and if i do function merge<x, y>(obj1: x, obj2: y): x & y { return { obj1, obj2 }; } why this give error give me breif indepth explantion

Understanding the `merge` Function with Generics

Let's break down both versions of the `merge` function to understand why one works and the other gives an error.

1. Correct Version: Using Spread Operator (...)

typescript

Copy

Download

```
function merge<X, Y>(obj1: X, obj2: Y): X & Y {
    return { ...obj1, ...obj2 }; // ✅ Correct
}

const merged = merge({ name: "Alice" }, { age: 30 });
// merged = { name: "Alice", age: 30 } (type: { name: string } & { age: number })
```

How It Works:

1. Generics (<x, y>)

- X and Y are **type variables** representing the types of obj1 and obj2.
- When you call `merge({ name: "Alice" }, { age: 30 })`, TypeScript infers:
 - X = { name: string }
 - Y = { age: number }

2. Return Type (x & y)

- X & Y means an **intersection type** (an object with both X and Y properties).
- So, the return type is { name: string } & { age: number } → { name: string; age: number }.

3. Spread Operator (...)

- { ...obj1, ...obj2 } **merges** the properties of obj1 and obj2 into a new object.
- Example:

js

Copy

Download

```
{ ...{ name: "Alice" }, ...{ age: 30 } }
// becomes { name: "Alice", age: 30 }
```

2. Incorrect Version: Without Spread ({ obj1, obj2 })

typescript

Copy

Download

```
function merge<X, Y>(obj1: X, obj2: Y): X & Y {
    return { obj1, obj2 }; // ❌ Error
}
```

Why This Fails:

1. Returned Structure is Different

- { obj1, obj2 } creates an object with **keys obj1 and obj2**, not a merged object.
- Example:

js

Copy

Download

```
{ obj1: { name: "Alice" }, obj2: { age: 30 } }
```

- This does **not** match the expected return type `X & Y({ name: string; age: number })`.

2. TypeScript Error

- The return type `X & Y` expects a **single object with all properties from X and Y**.
 - But `{ obj1, obj2 }` returns an object with **nested obj1 and obj2**, which is incompatible.
-

Fixing the Incorrect Version

If you really wanted to return `{ obj1, obj2 }`, you should change the return type:

typescript

Copy

Download

```
function merge<X, Y>(obj1: X, obj2: Y): { obj1: X; obj2: Y } {  
    return { obj1, obj2 }; // ✅ Now correct  
}  
  
const result = merge({ name: "Alice" }, { age: 30 });  
// result = { obj1: { name: "Alice" }, obj2: { age: 30 } }
```

But this is **not a merge**—it just wraps the objects inside another object.

Key Takeaways

Approach	Returned Value	Type
<code>{ ...obj1, ...obj2 }</code>	<code>{ name: "Alice", age: 30 }</code>	<code>X & Y (merged)</code>
<code>{ obj1, obj2 }</code>	<code>{ obj1: { name: "Alice" }, obj2: { age: 30 } }</code>	<code>{ obj1: X; obj2: Y } (nested)</code>

- **Spread (...)** merges objects into one.
- `{ obj1, obj2 }` nests them under new keys.
- **TypeScript enforces the return type strictly**, so mismatches cause errors.

Would you like more examples of merging with generics? 

```
function merge<X, Y>(obj1: X, obj2: Y): X & Y { return { obj1, obj2 }; } i mean if i call the merge like this  
merge<number, boolean>(89, true);
```

If you call the `merge` function like this:

typescript

Copy

Download

```
function merge<X, Y>(obj1: X, obj2: Y): X & Y {
    return { obj1, obj2 }; // ✗ Type Error
}

const result = merge<number, boolean>(89, true);
```

What Happens?

1. Expected Return Type (`X & Y`)

- Since `X = number` and `Y = boolean`, the expected return type is `number & boolean`.
- `number & boolean` is an **impossible type** because no value can be both a `number` and `boolean` at the same time (TypeScript treats this as `never`).

2. Actual Return Value (`{ obj1: number, obj2: boolean }`)

- The function returns `{ obj1: 89, obj2: true }`, which has the type:

typescript

Copy

Download

```
{ obj1: number; obj2: boolean }
```

- This **does not match** `X & Y (number & boolean)`, so TypeScript throws an error.

Why Does TypeScript Reject This?

- `X & Y` expects a single object that satisfies both `X` and `Y`.
 - For `number & boolean`, this is impossible (no value exists that is both a `number` and `boolean`).
 - `{ obj1, obj2 }` creates a new object with two properties (`obj1` and `obj2`), not a merged type.
-

How to Fix It?

Option 1: Use Spread (...) for Real Merging

typescript

Copy

Download

```
function merge<X, Y>(obj1: X, obj2: Y): X & Y {
    return { ...obj1, ...obj2 }; // ✓ Works if X and Y are objects
}

const result = merge({ a: 1 }, { b: true });
// result = { a: 1, b: true } (type: { a: number } & { b: boolean })
```

- **Works only if X and Y are objects.**
- **Fails for primitives (number, boolean, etc.)** because you can't spread them.

Option 2: Change Return Type to Reflect Actual Structure

typescript

[Copy](#)

[Download](#)

```
function merge<X, Y>(obj1: X, obj2: Y): { obj1: X; obj2: Y } {
    return { obj1, obj2 }; // ✅ Correct (but not a merge)
}

const result = merge<number, boolean>(89, true);
// result = { obj1: 89, obj2: true } (type: { obj1: number; obj2: boolean })
```

- This is not a merge, but it matches the actual return type.

Option 3: Use a Union Type ($X \mid Y$) Instead of Intersection ($X \& Y$)

typescript

[Copy](#)

[Download](#)

```
function merge<X, Y>(obj1: X, obj2: Y): X | Y {
    return Math.random() > 0.5 ? obj1 : obj2; // Returns either X or Y
}

const result = merge<number, boolean>(89, true);
// result = 89 or true (type: number | boolean)
```

- This works, but it doesn't merge anything—just returns one of the two values.

Key Takeaways

Approach	Return Type	Works For	Example Output
{ ...obj1, ...obj2 } $X \& Y$		Objects only	{ a: 1, b: true }
{ obj1, obj2 }	{ obj1: X; obj2: Y }	Any types	{ obj1: 89, obj2: true }
'X	'Y' (Union)	'X'	'Y'

- **X & Y (intersection) expects a merged type**, which only makes sense for objects.
- **Primitives (number, boolean) cannot be merged**—they can only be wrapped in another structure.
- **TypeScript enforces strict type safety**, so mismatches cause errors.

Would you like a real-world example where X & Y merging is useful? 

```
interface lengthy{ length: number, toUpperCase: () => string, } function getLength<x extends lengthy>(value:x):void {
console.log(value.length, value.toUpperCase()); } getLength("string"); i didn't understand that how is it working
because i have some doubts 1. how we are able to pass "string" when the type is lengthy which is an object of length
and toUpperCase function. 2. if we pass the "string" to the function how it is able to compute the length of the string
and toUppercase because i didn't even implement that function toUpperCase and length is not an even function
though. 3. why i am unable to pass the array, number, boolean. give full indepth reasoning and explanation
```

Deep Dive into `extends` Constraints and TypeScript Behavior

Let's break down your example step by step to understand why it works with `string` but fails with `number`, `boolean`, or even an array (unless properly structured).

1. The `lengthy` Interface

Your interface defines two properties:

typescript

Copy

Download

```
interface lengthy {  
    length: number;  
    toUpperCase: () => string;  
}
```

This means any type that satisfies `lengthy` must have:

- A `length` property (of type `number`).
 - A `toUpperCase` method (that returns a `string`).
-

2. How `string` Satisfies the `lengthy` Interface

When you pass a `string`:

typescript

Copy

Download

```
getLength("hello");
```

TypeScript **does not throw an error** because:

- `string` has a built-in `length` property (e.g., `"hello".length` returns 5).
- `string` has a built-in `toUpperCase()` method (e.g., `"hello".toUpperCase()` returns "HELLO").

Behind the Scenes: TypeScript's Structural Typing

TypeScript uses **structural typing** (duck typing), meaning:

"If it walks like a duck and quacks like a duck, it's a duck."

In this case:

- `string` already has `length` and `toUpperCase()`, so it matches the `lengthy` interface **implicitly**.
 - You didn't have to explicitly declare `string` implements `lengthy`.
-

3. Why `number`, `boolean`, and `Raw Array` Fail

Case 1: `number`

typescript

Copy

Download

```
notLength(12); // ❌ Error
```

```
getLength(42); // ✗ Error
```

- **Reason:**

- number does **not** have a `length` property (in JavaScript/TypeScript).
- number does **not** have a `toUpperCase()` method.
- Fails the `extends Lengthy` constraint.

Case 2: boolean

typescript

Copy

Download

```
getLength(true); // ✗ Error
```

- **Reason:**

- boolean has **no** `length` **or** `toUpperCase()`.
- Fails the constraint.

Case 3: Raw Array (Without `toUpperCase`)

typescript

Copy

Download

```
getLength([1, 2, 3]); // ✗ Error (unless modified)
```

- **Problem:**

- Arrays **do** have `length`, but they **do not** have `toUpperCase()`.
- Fails because `toUpperCase()` is missing.

Fixing the Array Case

If you define an array-like object with `toUpperCase`:

typescript

Copy

Download

```
const customArray = {
  length: 3,
  toUpperCase: () => "HELLO", // Manually added
  0: 1, 1: 2, 2: 3 // Array-like structure
};
getLength(customArray); // ✅ Works!
```

Now it matches `Lengthy` because:

- It has `length: number`.
- It has `toUpperCase(): string`.

4. Why You Didn't Need to Implement `length` or `toUpperCase`

- `string` is a primitive but has built-in methods (due to JavaScript's boxing behavior).
 - When you call `"hello".toUpperCase()`, JavaScript temporarily wraps the string in a `String` object, which has these methods.
 - TypeScript knows this and treats `string` as compatible with `{ length: number, toUpperCase(): string }`.
-

5. Key Takeaways

Type	Works with <code>getLength?</code>	Why?
<code>string</code>	✓ Yes	Has <code>length</code> and <code>toUpperCase()</code> .
<code>number</code>	✗ No	Missing both <code>length</code> and <code>toUpperCase()</code> .
<code>boolean</code>	✗ No	Missing both.
<code>Array</code> (normal)	✗ No	Has <code>length</code> but missing <code>toUpperCase()</code> .
Custom object with <code>length + toUpperCase()</code>	✓ Yes	Explicitly matches <code>lengthy</code> .

6. How to Make It Work with Arrays?

If you **only** care about `length` (and not `toUpperCase`), modify the interface:

typescript

Copy

Download

```
interface hasLength {
    length: number;
}

function getLength<T extends hasLength>(value: T): void {
    console.log(value.length);
}

getLength([1, 2, 3]); // ✓ Works (arrays have `length`)
getLength("hello"); // ✓ Works (strings have `length`)
getLength({ length: 5 }); // ✓ Works (custom object)
// getLength(42); // ✗ Fails (numbers don't have `length`)
```

Summary

1. `string` works because it **implicitly matches** the `lengthy` interface (built-in props/methods).
2. `number`, `boolean`, and raw arrays fail because they **lack required properties**.
3. **TypeScript's structural typing** allows implicit matching—no need for explicit `implements`.
4. **You can make it work with arrays** by defining a proper object or relaxing the interface.

Would you like more examples of constrained generics? 

```
class Store<T> { private state: T; constructor(initialState: T) { this.state = initialState; } getState(): T { return this.state; }
update(partial: Partial<T>) { this.state = { ...this.state, ...partial }; } } const userStore = new Store({ name: "", age:
```

0 })userStore.update({ name: "Charlie" }); console.log(userStore.getState()); i just not understood the work of Partial<T> and i didn't defined it and can use it without defining it how and how is it working what is in it explain it deeply with proper explantion

Deep Dive into Partial<T> in TypeScript

Let's break down how `Partial<T>` works in your `Store` class example, why you can use it without defining it, and how it enables partial updates to the state.

1. What is `Partial<T>`?

`Partial<T>` is a **built-in utility type** in TypeScript that makes all properties of `T` optional.

Definition (From TypeScript Docs):

typescript

Copy

Download

```
type Partial<T> = {  
  [P in keyof T]?: T[P];  
};
```

- It takes a type `T` and creates a new type where **all properties are optional** (`? modifier`).
 - This means you can pass **only some properties** of `T` instead of all.
-

2. How `Partial<T>` Works in Your Store Class

Original Class Definition

typescript

Copy

Download

```
class Store<T> {  
  private state: T;  
  constructor(initialState: T) {  
    this.state = initialState;  
  }  
  getState(): T {  
    return this.state;  
  }  
  update(partial: Partial<T>) {  
    this.state = { ...this.state, ...partial }; // Merges old & new state  
  }  
}
```

Example Usage

typescript

Copy

Download

```
const userStore = new Store({ name: "", age: 0 }); // T = { name: string; age: number }
userStore.update({ name: "Charlie" }); // ✅ Works (only `name` is provided)
```

What Happens Under the Hood?

1. Initial State (T):

- T is inferred as { name: string; age: number }.

2. Partial<T> in update():

- Partial<T> becomes:

typescript

Copy

Download

```
{ name?: string; age?: number }
```

- This means update() accepts:

- { name: "Charlie" } (only name),
- { age: 30 } (only age),
- { name: "Charlie", age: 30 } (both),
- {} (nothing, though it won't change the state).

3. Merging with Spread (...)

- { ...this.state, ...partial } combines:

- The **current state** ({ name: "", age: 0 }),
- The **partial update** ({ name: "Charlie" }).

- Result:

typescript

Copy

Download

```
{ name: "Charlie", age: 0 } // `name` updated, `age` remains
```

3. Why Can You Use Partial<T> Without Defining It?

- Partial<T> is a **built-in utility type** (like Pick<T>, Omit<T>, Record<K, V>).
- It's globally available in TypeScript—no need to import or define it.

Other Built-in Utility Types

Utility Type	Purpose	Example
Partial<T>	Makes all properties optional	Partial<{ a: number }> → { a?: number }
Required<T>	Makes all properties required	Required<{ a?: number }> → { a: number }
Pick<T, K>	Picks specific properties	Pick<{ a: number, b: string }, 'a'> → { a: number }

`Omit<T, K>` Omits specific properties `Omit<{ a: number, b: string }, 'a'> → { b: string }`

4. What If `Partial<T>` Didn't Exist?

Without `Partial<T>`, you'd have to:

1. **Pass the full state every time** (annoying):

typescript

Copy

Download

```
update(newState: T) {
    this.state = newState; // Must provide all properties
}
```

- Would force:

typescript

Copy

Download

```
userStore.update({ name: "Charlie", age: 0 }); // Must repeat `age`
```

2. **Manually define optional properties** (messy):

typescript

Copy

Download

```
update(partial: { name?: string; age?: number }) {
    this.state = { ...this.state, ...partial };
}
```

- Not reusable—must rewrite for every `T`.

`Partial<T>` solves this elegantly!

5. Advanced: How `Partial<T>` is Implemented

If we were to write `Partial` ourselves:

typescript

Copy

Download

```
type MyPartial<T> = {
    [K in keyof T]?: T[K];
};
```

- `keyof T` = Gets all keys of `T` (e.g., `"name"` | `"age"`).
- `[K in keyof T]?: T[K]` = Makes each property optional.

Example Breakdown

```
For T = { name: string; age: number }:
```

typescript

Copy

Download

```
type PartialUser = MyPartial<{ name: string; age: number }>;
// Equivalent to:
type PartialUser = {
  name?: string;
  age?: number;
};
```

6. When to Use `Partial<T>`

- **Form Updates:** Allow submitting only changed fields.
 - **API PATCH Requests:** Send partial updates to a server.
 - **State Management:** Merge partial state (like Redux or React `useState`).
-

7. Key Takeaways

1. `Partial<T>` makes all properties optional.
 2. It's a built-in utility type—no need to define it.
 3. Works with spread (...) to merge partial updates.
 4. Without it, you'd manually handle optional props (messy!).
 5. Used heavily in state management (React, Redux, Vuex).
-

Final Example: `Partial<T>` in API Calls

typescript

Copy

Download

```
async function updateUser(id: string, changes: Partial<User>) {
  await fetch(`/users/${id}`, {
    method: "PATCH",
    body: JSON.stringify(changes),
  });
}

// Only update the email:
updateUser("123", { email: "new@example.com" });
```

This is why `Partial<T>` is powerful and widely used in TypeScript! 

```
// async function fetchData<T>(url: string): Promise<T> { // const response = await fetch(url); // return response.json();
// } // interface appUser { name: string; } // const user = await fetchData<appUser>("/api/user"); in this example the
return type is Promise do we can return a promise ? and give me all return types and input types in typescript
```

Understanding Return Types and Input Types in TypeScript

In TypeScript, functions can return different types of values, including **Promises**, and they can accept various input types. Let's break down your example and explore all possible return and input types.

1. Your Example: `fetchData` with Generics and Promises

typescript

Copy

Download

```
async function fetchData<T>(url: string): Promise<T> {
    const response = await fetch(url);
    return response.json(); // Returns a Promise<T>
}

interface AppUser { name: string; }

// Usage
const user = await fetchData<AppUser>("/api/user"); // user is of type `AppUser`
```

Key Observations:

1. Return Type (`Promise<T>`):

- `fetchData` is an `async` function, so it **always returns a Promise**.
- The resolved value of the `Promise` is of type `T` (inferred or explicitly set).
- `response.json()` also returns a `Promise`, which is unwrapped by `await`.

2. Input Type (`url: string`):

- The function takes a `string` input (`url`).

3. Generic Type (`<T>`):

- Allows flexibility in defining the expected return type (`AppUser` in this case).

2. All Possible Return Types in TypeScript

A function in TypeScript can return any valid type, including:

Return Type	Example	Description
Primitives (<code>number, string, boolean</code>)	<code>() => number</code>	Returns a basic type.
Objects (<code>{}, interface, type</code>)	<code>() => { name: string }</code>	Returns an object.
Arrays (<code>T[]</code>)	<code>() => string[]</code>	Returns an array.
Tuples (<code>[T, U]</code>)	<code>() => [string, number]</code>	Fixed-length array with typed positions.
Functions (<code>(() => void)</code>)	<code>() => () => void</code>	Returns another function.
<code>void</code>	<code>() => void</code>	No return value (like <code>console.log</code>).
	<code>..</code>	Function never returns (throws error)

<code>never</code>	<code>() => never</code>	or infinite loop).	
<code>any</code>	<code>() => any</code>	Opts out of type checking (avoid if possible).	
<code>unknown</code>	<code>() => unknown</code>	Safer alternative to <code>any</code> (requires type checking).	
<code>Promise<T></code>	<code>async () => T</code>	Returns a Promise resolving to <code>T</code> .	
Union Types (<code>'T</code> <code>U`</code>)	<code>U`</code>	<code>`() => string</code>	<code>number`</code> Returns one of multiple types.
Intersection Types (<code>T & U</code>)	<code>() => A & B</code>	Returns a combined type.	

3. All Possible Input (Parameter) Types in TypeScript

Functions can accept any valid type as input:

Input Type	Example	Description	
Primitives (<code>number, string, boolean</code>)	<code>(x: number) => void</code>	Accepts basic types.	
Objects (<code>{}, interface, type</code>)	<code>(user: { name: string }) => void</code>	Accepts an object.	
Arrays (<code>T[]</code>)	<code>(nums: number[]) => void</code>	Accepts an array.	
Tuples (<code>[T, U]</code>)	<code>(pair: [string, number]) => void</code>	Fixed-length array.	
Functions (<code>() => void</code>)	<code>(callback: () => void) => void</code>	Accepts a function.	
Optional Params (<code>?</code>)	<code>(x?: number) => void</code>	Parameter is optional.	
Rest Params (<code>...args</code>)	<code>(...args: number[]) => void</code>	Accepts variable arguments.	
Generics (<code><T></code>)	<code><T>(arg: T) => T</code>	Works with any type.	
Union Types (<code>'T</code> <code>U`</code>)	<code>U`</code>	<code>`(input: string</code>	<code>number` =></code> Accepts multiple types.
Intersection Types (<code>T & U</code>)	<code>(obj: A & B) => void</code>	Requires combined type.	<code>void`</code>

4. Deep Dive: How `Promise<T>` Works

Async Functions Always Return Promises

typescript

Copy

Download

```
async function getNumber(): Promise<number> {
    return 42; // Automatically wrapped in a Promise
}

// Equivalent to:
function getNumberManual(): Promise<number> {
    return Promise.resolve(42);
}
```

Await Unwraps Promises

typescript

typescript

Copy

Download

```
const numPromise: Promise<number> = getNumber();
const num: number = await numPromise; // Unwraps Promise<number> → number
```

Error Handling with try/catch

typescript

Copy

Download

```
try {
  const user = await fetchData<AppUser>("/api/user");
} catch (error) {
  console.error("Failed to fetch:", error);
}
```

5. Common Patterns with Return/Input Types

1. Returning Multiple Types (Union Types)

typescript

Copy

Download

```
function getValue(): string | number {
  return Math.random() > 0.5 ? "hello" : 42;
}
```

2. Accepting Callbacks (Higher-Order Functions)

typescript

Copy

Download

```
function measureTime(callback: () => void): number {
  const start = Date.now();
  callback();
  return Date.now() - start;
}
```

3. Generic Functions (Flexible Input/Output)

typescript

Copy

Download

```
function identity<T>(arg: T): T {
  return arg;
}
const str = identity("hello"); // T = string
const num = identity(42);    // T = number
```

4. Rest Parameters (Variable Arguments)

typescript

Copy

Download

```
function sum(...numbers: number[]): number {
    return numbers.reduce((a, b) => a + b, 0);
}
sum(1, 2, 3); // 6
```

6. Key Takeaways

1. **async functions always return `Promise<T>`.**
 2. **`Promise<T>` can be unwrapped using `await`.**
 3. **Input and return types can be any valid TypeScript type** (primitives, objects, functions, etc.).
 4. **Generics (`<T>`) make functions reusable with different types.**
 5. **Union (`|`) and intersection (`&`) types allow flexible typing.**
 6. **`void` means "no return," `never` means "never returns" (throws or infinite loop).**
-

Final Example: Full-Type-Safe API Fetch

typescript

Copy

Download

```
interface Post {
    id: number;
    title: string;
}

async function fetchPost(id: number): Promise<Post> {
    const response = await fetch(`/posts/${id}`);
    if (!response.ok) throw new Error("Failed to fetch");
    return response.json() as Promise<Post>; // Type assertion
}

// Usage
const post = await fetchPost(1); // post: Post
```

This ensures **end-to-end type safety** in API calls! 

Partial<T>, Required<T>, Pick<T>, Omit<T>, Record<K, V> give me very simple examples of these built-in utility types

Simple Examples of TypeScript Utility Types

Here are easy-to-understand examples of TypeScript's built-in utility types:

1. Partial<T>

Makes all properties of T optional.
(Useful for partial updates, like `setState` in React.)

Example

typescript

Copy

Download

```
interface User {  
    name: string;  
    age: number;  
}  
  
// All properties become optional  
type PartialUser = Partial<User>;  
/* Equivalent to:  
 {  
     name?: string;  
     age?: number;  
 }  
*/  
  
const updateUser: PartialUser = { name: "Alice" }; // ✅ No error (age is optional)
```

2. Required<T>

Makes all properties of T required.
(Opposite of `Partial`.)

Example

typescript

Copy

Download

```
interface Config {  
    id?: number;  
    mode?: string;  
}  
  
// All properties become required  
type RequiredConfig = Required<Config>;  
/* Equivalent to:  
 {  
     id: number;  
     mode: string;  
 }  
*/  
  
const strictConfig: RequiredConfig = { id: 1, mode: "dark" }; // ✅  
// const invalidConfig: RequiredConfig = { id: 1 }; // ❌ Error (missing `mode`)
```

3. Pick<T, K>

Picks specific properties K from T.
(Useful when you only need a subset of properties.)

Example

typescript

Copy

Download

```
interface Book {
  title: string;
  author: string;
  pages: number;
}

// Only pick 'title' and 'author'
type BookPreview = Pick<Book, "title" | "author">;
/* Equivalent to:
{
  title: string;
  author: string;
}
*/

const preview: BookPreview = { title: "TS Guide", author: "Alice" }; // ✅
// const invalid: BookPreview = { title: "TS Guide" }; // ❌ (missing `author`)
```

4. Omit<T, K>

Removes specific properties K from T.
(Opposite of `Pick`.)

Example

typescript

Copy

Download

```
interface Movie {
  title: string;
  director: string;
  year: number;
}

// Remove 'year'
type MovieWithoutYear = Omit<Movie, "year">;
/* Equivalent to:
{
  title: string;
  director: string;
}
*/

const movie: MovieWithoutYear = { title: "Inception", director: "Nolan" }; // ✅
// const invalid: MovieWithoutYear = { title: "Inception", year: 2010 }; // ❌
```

5. Record<K, V>

Creates an object type with keys K and values V.
(Useful for dictionaries.)

Example

typescript

Copy

Download

```
type Weekday = "Mon" | "Tue" | "Wed";
type Schedule = Record<Weekday, string>;
/* Equivalent to:
   {
     Mon: string;
     Tue: string;
     Wed: string;
   }
*/
const schedule: Schedule = {
  Mon: "Work",
  Tue: "Gym",
  Wed: "Meeting",
}; // ✅
// const invalid: Schedule = { Mon: "Work" }; // ❌ (missing Tue/Wed)
```

Key Takeaways

Utility Type	Purpose	Example
Partial<T>	Makes all properties optional	Partial<User> → { name?: string; age?: number }
Required<T>	Makes all properties required	Required<Config> → { id: number; mode: string }
Pick<T, K>	Picks specific properties	Pick<Book, "title"> → { title: string }
Omit<T, K>	Removes specific properties	Omit<Movie, "year"> → { title: string; director: string }
Record<K, V>	Maps keys to values	`Record<"A" "B", number>→{ A: number; B: number }`

These utilities help you **avoid manual type definitions** and keep your code DRY (Don't Repeat Yourself)! 🚀