**✅ Chapter 16: Type Inference Deep Dive (infer, Contextual Typing)**

**📘 1. Concept Overview**

TypeScript automatically infers types when you don’t explicitly declare them — this is called **Type Inference**.  
But with **infer**, you take **control and extract** a type from within another type.

**🧠 infer lets you:**

* Extract return types, argument types, property types, etc.
* Create dynamic conditional types
* Build advanced reusable utilities

**🧪 2. Basic Syntax of infer**

type GetReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

👆 If T is(extends) a function type((...args: any[]) => infer R ), extract the return type into R, otherwise return never.

**📦 Example:**

type ExampleFn = () => string;

type Result = GetReturnType<ExampleFn>;

// ✅ Result = string

**🔍 3. Use Cases of infer**

| **Use Case** | **Example Utility** |
| --- | --- |
| Extract function return type | ReturnType<T> |
| Extract function argument type | Parameters<T> |
| Extract resolved type from Promise | Awaited<T> or UnwrapPromise<T> |
| Extract value from object | ValueOf<T> |
| Custom infer for nested types | Flatten<T> or First<T> |

**🧪 4. Examples for Each Use Case**

**✅ 4.1 Extract Return Type of a Function**

ts

CopyEdit

type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

type Fn = () => number;

type A = ReturnType<Fn>; // ✅ A = number

**✅ 4.2 Extract Argument Types**

ts

CopyEdit

type Parameters<T> = T extends (...args: infer P) => any ? P : never;

type Fn = (a: number, b: string) => void;

type B = Parameters<Fn>; // ✅ B = [number, string]

**✅ 4.3 Extract Resolved Type from a Promise**

type UnwrapPromise<T> = T extends Promise<infer U> ? U : T;

type C = UnwrapPromise<Promise<string>>; // ✅ C = string

**✅ 4.4 Extract Value Types from Objects**

ts

CopyEdit

type ValueOf<T> = T extends Record<string, infer V> ? V : never;

type D = ValueOf<{ name: string; age: number }>;

// ✅ D = string | number

**✅ 4.5 Flatten Array  
  
type Flatten<T> = T extends (infer U)[] ? U : T;  
or**

type Flatten<T> = T extends Array<infer U> ? U : T;

type E = Flatten<string[]>; // ✅ E = string

type F = Flatten<number>; // ✅ F = number

**💬 5. Contextual Typing**

TypeScript infers types **based on surrounding context**:

// ✅ 1️⃣ Variable Assignment — inferred from array elements

const nums = [1, 2, 3];

// nums: number[]

// ✅ 2️⃣ Function Parameter with Default — inferred as string

const greet = (name = "Likan") => name.toUpperCase();

// name: string

// ✅ 3️⃣ Event Listener — inferred from DOM context

document.addEventListener("click", (e) => {

// e: MouseEvent (inferred from "click" event)

e.preventDefault();

});

// ✅ 4️⃣ Promise Callback — inferred from fetch() return type

fetch("/api/data").then((res) => {

// res: Response (inferred by TS from fetch API)

return res.json();

});

// ✅ 5️⃣ React JSX Callback — inferred from props or event type

type ButtonProps = {

onClick: () => void;

};

const Button = ({ onClick }: ButtonProps) => (

<button

onClick={(e) => {

// e: React.MouseEvent<HTMLButtonElement>

onClick();

}}

>

Click

</button>

);

// ✅ 6️⃣ Array.map — contextual typing from array type

const names = ["Likan", "Amar"];

const uppercased = names.map((name) => name.toUpperCase());

// name: string

// ✅ 7️⃣ Destructuring — inferred from object shape

const user = {

name: "Likan",

age: 30,

};

const { name, age } = user;

// name: string, age: number

// ✅ 8️⃣ Return type inferred from implementation

const double = (n: number) => n \* 2;

// return type: number (contextually inferred)

// BONUS ✅ 9️⃣ setTimeout callback — parameter inferred as number (timer ID)

setTimeout((id) => {

// id: number (NodeJS.Timeout in Node, number in browser)

console.log("Timeout executed with ID:", id);

}, 1000);

// BONUS ✅ 🔟 Custom Type Inference Example (to contrast with infer)

type Flatten<T> = T extends (infer U)[] ? U : T;

type A = Flatten<string[]>; // string

type B = Flatten<number>; // number

But infer lets **you extract types from structures yourself**, not just rely on the compiler.

**📌 Summary (One-Liners)**

* infer is used in conditional types to **extract types** like return type, parameter type, etc.
* infer R or infer U gives a **temporary name** to the extracted type
* Common in built-in types like ReturnType, Parameters, InstanceType Helps you **build dynamic, reusable** type helpers

## 🔎 1. ReturnType<T> — Extracts return type of a function

### ✅ Built-in behavior:

ts

CopyEdit

type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;

### 🔹 Example:

ts

CopyEdit

function greet(name: string) {

return `Hello, ${name}`;

}

type GreetReturn = ReturnType<typeof greet>;

// GreetReturn = string ✅

🧠 infer R extracts whatever is **returned by the function**.

## 🔎 2. Parameters<T> — Extracts parameters of a function as tuple

### ✅ Built-in behavior:

ts

CopyEdit

type Parameters<T> = T extends (...args: infer P) => any ? P : never;

### 🔹 Example:

ts

CopyEdit

function sum(a: number, b: number): number {

return a + b;

}

type SumParams = Parameters<typeof sum>;

// SumParams = [number, number] ✅

🧠 infer P extracts the entire parameter list as a tuple.

## 🔎 3. InstanceType<T> — Extracts instance type from a class constructor

// ✅ Custom utility type that extracts the instance type from a class constructor

type MyInstanceType<T> =

T extends new (...args: any[]) => infer R ? R : never;

// ✅ Sample class

class User {

name: string = "Likan";

age: number = 30;

greet() {

return `Hello, I'm ${this.name}`;

}

}

// ✅ Usage — pass the constructor type using `typeof User`

type UserInstance = MyInstanceType<typeof User>;

/\*

Internally resolves to:

type UserInstance = {

name: string;

age: number;

greet(): string;

}

\*/

const user: UserInstance = new User(); // ✅ Correct usage

// ❌ Wrong usage — passing non-constructor type

type NotAConstructor = MyInstanceType<string>; // ❌ Fallbacks to never

// ✅ Another class

class Product {

id: number;

title: string;

constructor(id: number, title: string) {

this.id = id;

this.title = title;

}

}

type ProductInstance = MyInstanceType<typeof Product>;

const p: ProductInstance = new Product(1, "Phone");

// ✅ p is inferred as { id: number; title: string }

========

type MyInstanceType<T> =

T extends new (...args: any[]) => infer R ? R : never;

🧠 Breakdown:

T must be a constructor function → new (...args: any[]) => ...

infer R extracts the type of the value returned when you call new T()

In TypeScript, when you use new, it creates a class instance

So R becomes the instance type of the class

### 📝 Note: InstanceType Syntax Is Like ReturnType — Just with new

The structure of InstanceType<T> is almost the same as ReturnType<T>,  
except that it matches **class constructors** instead of functions —  
so it uses new (...args) instead of just (...args).

// ✅ A class (returns instance when used with `new`)

class User {

name = "Likan";

age = 30;

}

// ✅ `typeof User` gives constructor type: new () => User

// ✅ `InstanceType<typeof User>` extracts the instance type using `infer R`

type Extracted = InstanceType<typeof User>;

// 🔍 Extracted = { name: string; age: number }

const u: Extracted = new User(); // ✅ valid instance

### 📝 Note: InstanceType works with ****classes****, not plain objects

We use typeof ClassName to pass the **constructor function** to InstanceType.

* ✅ If you're using a **class**, typeof User gives new () => User
* ❌ If you're using a **plain object**, typeof just gives the **object's shape**, not a constructor — so InstanceType<typeof obj> ❌ won't work

**🧠 Interview Q&A**

**Q1. What does the infer keyword do in TypeScript?**  
✅ It extracts a type from within a structure (like function return types, array elements, etc.) during conditional type evaluation.

**Q2. Where is infer used in real-world code?**  
✅ Used in utilities like ReturnType<T>, Parameters<T>, or custom types like UnwrapPromise<T>, Flatten<T>, etc.

**Q3. Can you use infer outside a conditional type?**  
❌ No. infer only works **inside** a T extends X ? ... : ... conditional expression.

**Q4. What happens if the condition fails (not matched)?**  
✅ The : never fallback is used.