# The `infer` Keyword

The lesson goes through some of the remaining built-in conditional types and introduces the `infer` keyword.

WE'LL COVER THE FOLLOWING



## Parameters

The `Parameters` type takes a function type and returns a tuple type representing types of all parameters of the function. Sounds magical, doesn't it?

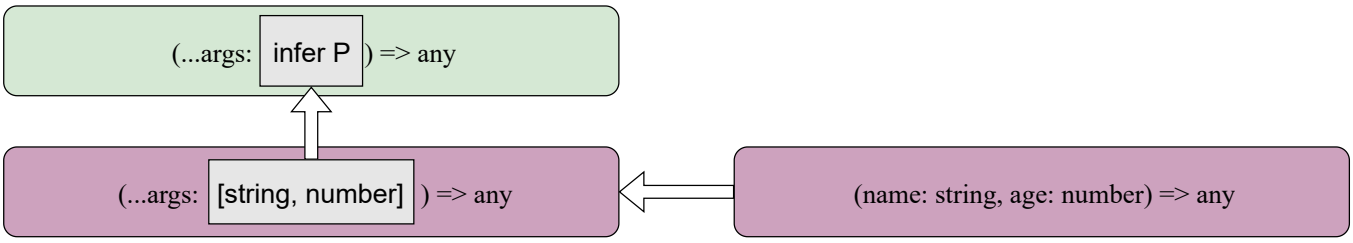
```
type Parameters<T extends (...args: any) => any> = T extends (...args: infer P) => any ? P :

const sayHello = (name: string, age: number) => `Hello ${name}, your age is ${age}`;
type SayHelloParams = Parameters<typeof sayHello>; // [string, number]
```

Hover over `SayHelloParams` to see the inferred type.

Let's break down this definition. First, the `Parameters` type has a type argument constraint that says that `T` must be a function type (`(...args: any) => any`). Next, inside the conditional expression we say that `T extends (...args: infer P) => any`. `P` is not a type argument, it's like a **type variable** which will hold the actual type of function parameters (as a tuple). The `infer` keyword lets us *unwrap* a type argument and pick a consistent part.

Another way to think about this is that `T` is matched against the `(...args: infer P) => any` *pattern*. If it matches the pattern, then `P` becomes the type that is at the same place in `T` as `infer P` in the pattern. For example, let's assume that `T` is `(name: string, age: number) => string`. It gets matched against `(...args: infer P) => any`. Since it can also be represented as `(...args: [string, number]) => string`, it matches the pattern. The

equivalent of `infer P` in `T` is `[string, number]`, so `P` gets resolved to `[string, number]` inside the positive branch of the conditional expression.




Symbolic illustration of how the type (in purple) is matched against the "pattern" (in green).

Having `P` contain the type we're interested in, we simply return it in the positive branch of the condition. We return `never` in the negative branch; we'll never enter this branch, as guaranteed by the constraint on `T`.

## Return #

`ReturnType` works in a similar way. It accepts a function type, `T`, and it extracts the return type of the function using the `infer` keyword. This time `infer R` is situated in another place of the pattern. When `T` is matched against the pattern, `R` gets resolved to the return type of the function.

```
type ReturnType<T extends (...args: any) => any> = T extends (...args: any) => infer R ? R :

const sayHello = (name: string, age: number) => `Hello ${name}, your age is ${age}`;
type SayHelloReturnType = ReturnType<typeof sayHello>; // string
```

Hover over `SayHelloReturnType` to see the inferred type.

One example of where `ReturnType` is useful is in Redux where you define action creators and reducers. A reducer accepts a state object and an action object. You can use `ReturnType` of the action creator to type the action object.

```
function fetchDataSuccess(data: string[]) {
  return {
    type: 'fetchDataSuccess',
    payload: data
  }
}

function reduceFetchDataSuccess(
   state: State,
   { payload }: ReturnType<typeof fetchDataSuccess>
```

```
) {}
```

Hover over `payload` to see the inferred type.

The next lesson walks through an example of using conditional types in practice.