Generic Inference

This lesson explains how TypeScript can infer type.

WE'LL COVER THE FOLLOWING

Inference and generic

Inference with generic is possible. If a function takes a parameter of type `T` and it returns `T` as well, the parameter assigned will define the generic type, and the return is inferred to be type `T`. The only exception is that if your function doesn't use the value `T`, it will return an empty object type.

```
function genericInferred<T>(param: T) {}
genericInferred("str"); // T is of type string by inference
genericInferred<string>("str"); // Same as above, no inference
```

The function above, in the example, is taking a string at **line 2** implicitly and at line **3** explicitly but in both cases could be replaced with any other type. The reason is that `T` does not have any constraint.

```
function genericInferred<T>(param: T) {}
genericInferred(1);
genericInferred(true);
genericInferred({ custom: "sure" });
```

Implicit type with generic

The type does not need to be specified, but regardless, the type passed inside the function will always be generic. For example, **line 2** is using a number, **line 3** is using a boolean and **line 4** is using a custom object.

As seen, we need to use generic constraints if we want to be able to access a portion of the inferred type.

```
function genericInferred<T extends string>(param: T) {
    return param.length;
}
console.log(genericInferred("Four"));
// genericInferred(123); // Does not transpile
type UUID = string;
let id: UUID = "123-456";
console.log(genericInferred(id));
```

The code above illustrates that the generic function takes anything that extends string. **Line 1** has the constraint with the `extends` after the `T`.

This could be a string or a type based on a string (like the `UUID` type). At **line 6**, we define a custom `UUID` type. It is actually an alias to `string`. While it might insignificant, it is more describing and easier to understand what the string should contain.

In both cases, the code infers the type and the specific type of `T` is not explicitly needed. On the opposite, the commented **line 5** cause a transpilation error because the value `123` is of type `number` and does not fulfill the constraint that `T` ought to be extending `string`.