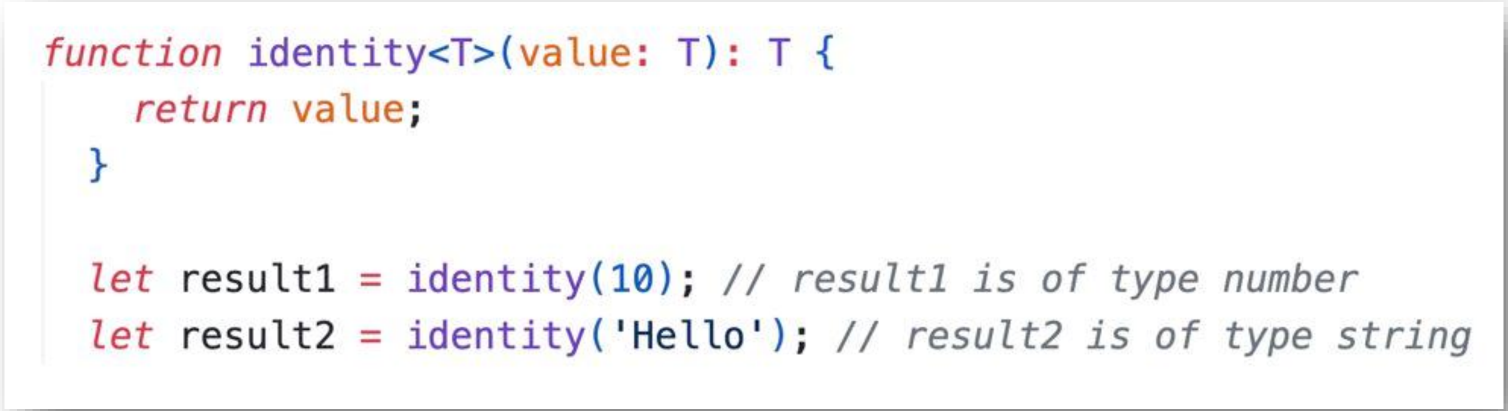
Lecture 11: TypeScript Generics and Advanced Types

What are Generics?

- **Generics** allow you to write flexible and reusable code by enabling types to be specified later, not upfront.

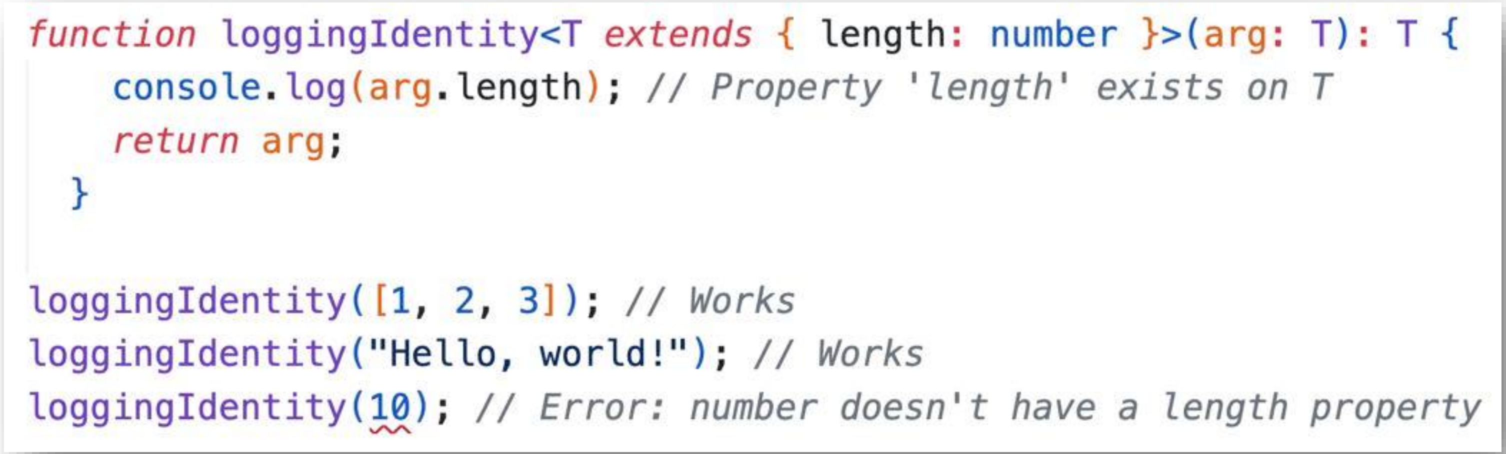
- Prevent redundancy by defining a function or class that works with any data type



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

  let result1 = identity(10); // result1 is of type number
  let result2 = identity('Hello'); // result2 is of type string
```

Generic Constraints

- Restrict the types that can be used as generic type arguments
- **T extends { length: number }** means **T** must have a **length** property



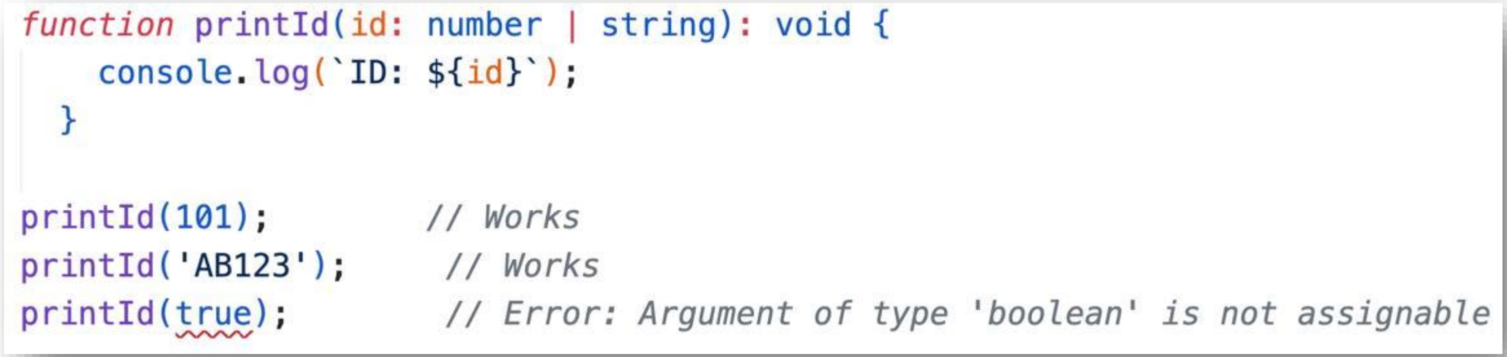
```
function loggingIdentity<T extends { length: number }>(arg: T): T {
    console.log(arg.length); // Property 'length' exists on T
    return arg;
  }

loggingIdentity([1, 2, 3]); // Works
loggingIdentity("Hello, world!"); // Works
loggingIdentity(10); // Error: number doesn't have a length property
```

Advanced Types

- Union Types
- Intersection Types
- Conditional Types

Union Types

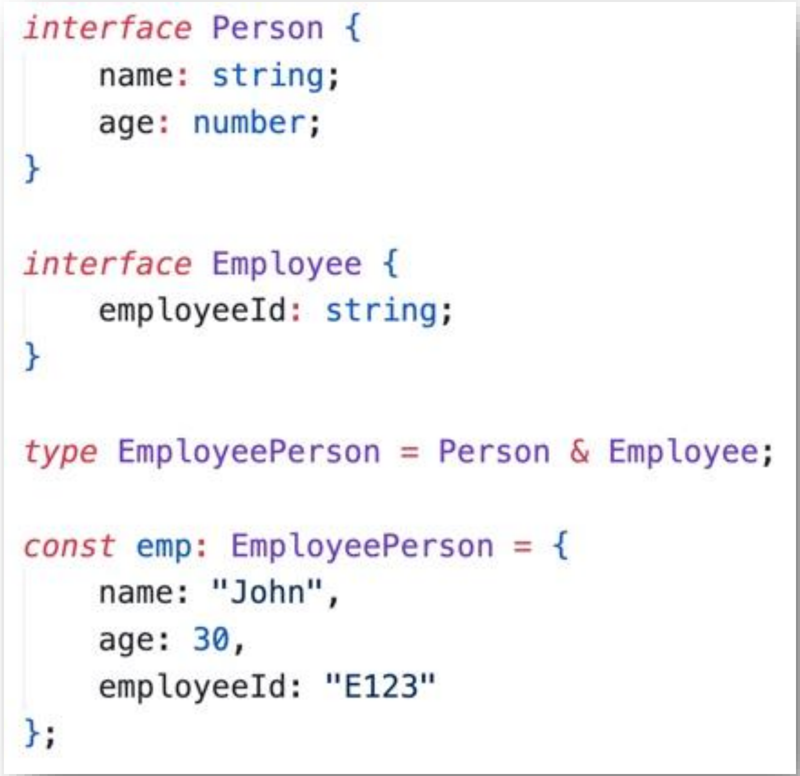
A **union type** allows a variable to be one of several types



```
function printId(id: number | string): void {
    console.log(`ID: ${id}`);
  }

printId(101);         // Works
printId('AB123');      // Works
printId(true);         // Error: Argument of type 'boolean' is not assignable
```

Intersection Types

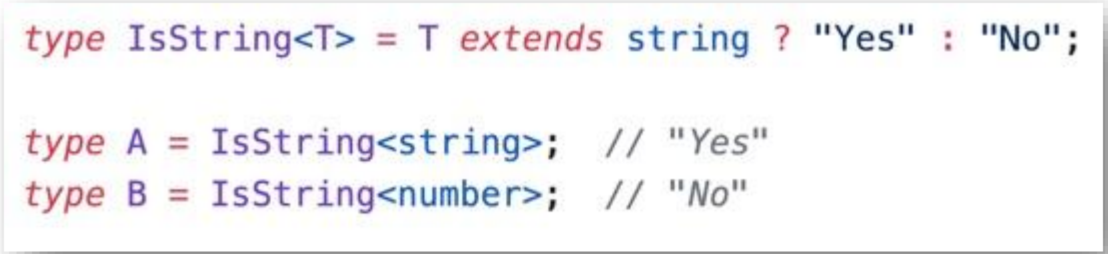
combines multiple types into one



```
interface Person {
    name: string;
    age: number;
}

interface Employee {
    employeeId: string;
}

type EmployeePerson = Person & Employee;

const emp: EmployeePerson = {
    name: "John",
    age: 30,
    employeeId: "E123"
};
```

Conditional Types

A **conditional type** provides a way to define a type based on a condition

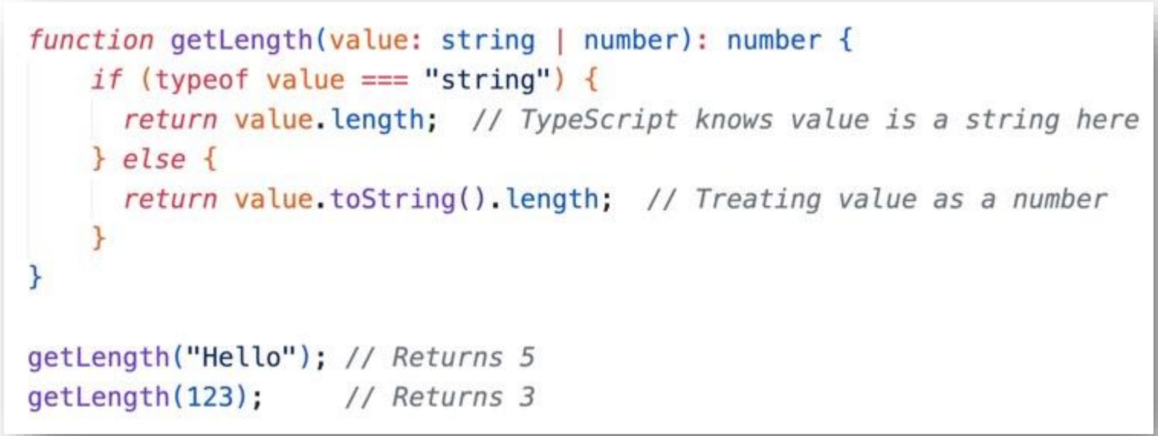


```
type IsString<T> = T extends string ? "Yes" : "No";

type A = IsString<string>;  // "Yes"
type B = IsString<number>;  // "No"
```

Type Guards

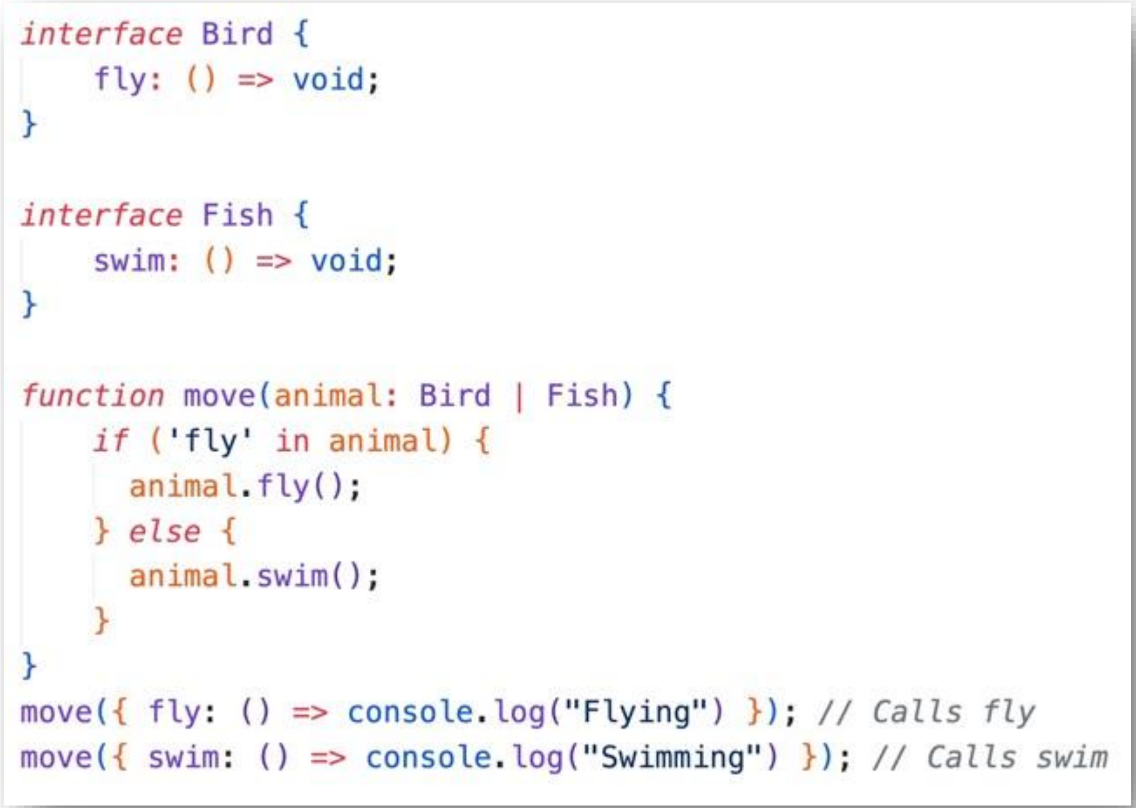
Type Guards are expressions that narrow down the type of a variable within a condition block

typeof narrows the type to either **string** or **number**



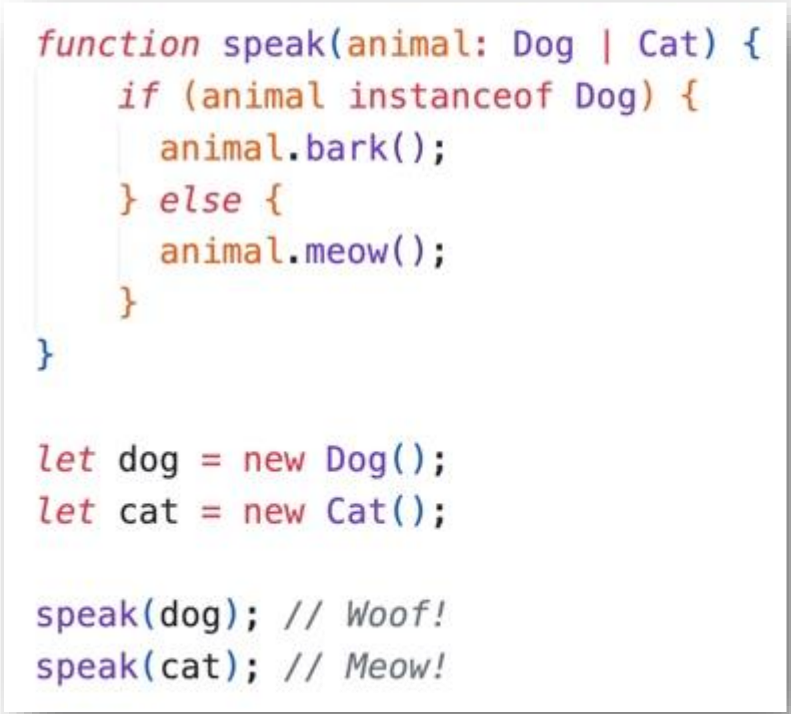
```
function getLength(value: string | number): number {
    if (typeof value === "string") {
      return value.length;  // TypeScript knows value is a string here
    } else {
      return value.toString().length;  // Treating value as a number
    }
}

getLength("Hello"); // Returns 5
getLength(123);     // Returns 3
```

Use **in** to check if property exists on an object, narrowing the type.



```
interface Bird {
    fly: () => void;
}

interface Fish {
    swim: () => void;
}

function move(animal: Bird | Fish) {
    if ('fly' in animal) {
      animal.fly();
    } else {
      animal.swim();
    }
}
move({ fly: () => console.log("Flying") }); // Calls fly
move({ swim: () => console.log("Swimming") }); // Calls swim
```

instanceof narrows the type by checking if an object is an instance of a class



```
function speak(animal: Dog | Cat) {
    if (animal instanceof Dog) {
      animal.bark();
    } else {
      animal.meow();
    }
}

let dog = new Dog();
let cat = new Cat();

speak(dog); // Woof!
speak(cat); // Meow!
```