Training TypeScript

Module: Type Guards

Peter Kassenaar

info@kassenaar.com

"Type Guards allow you to narrow down the type of an object within a conditional block"

Two main types of guards:

- `typeof` operator (…/`18-type-guard-typeof.ts`)
- `instanceof` operator (…/`19-type-guard-instanceof.ts`)

If you want to – you can declare your own, custom Type Guards (`20-type-guard-user-defined.ts`)

Type Guard using `typeof` operator

```
// type-guard-typeof.ts
function foo(bar: string | number) {
  if (typeof bar === 'string') {
    // do something, we KNOW it is a 'string' value
    // For instance, we get intellisense on all string methods
    return bar.toUpperCase();
  }
  // HERE, TypeScript KNOWS it should be a number value,
  // because we handled the string value above
  return bar.toFixed(2);
}
```

More real life example – step 1

```
class Employee {
  constructor(public name: string, public age: string | number) {}
}

function getEmployeeAge(employee: Employee) {
  // HERE, we implement the Type Guard
  /////……
}

const employeeAgeFromString = getEmployeeAge(
  new Employee('Dirk', '29')
);

console.log(employeeAgeFromString);
```

Step 2 – implement the type guard

We want the age of an employee ALWAYS to be of type `number`

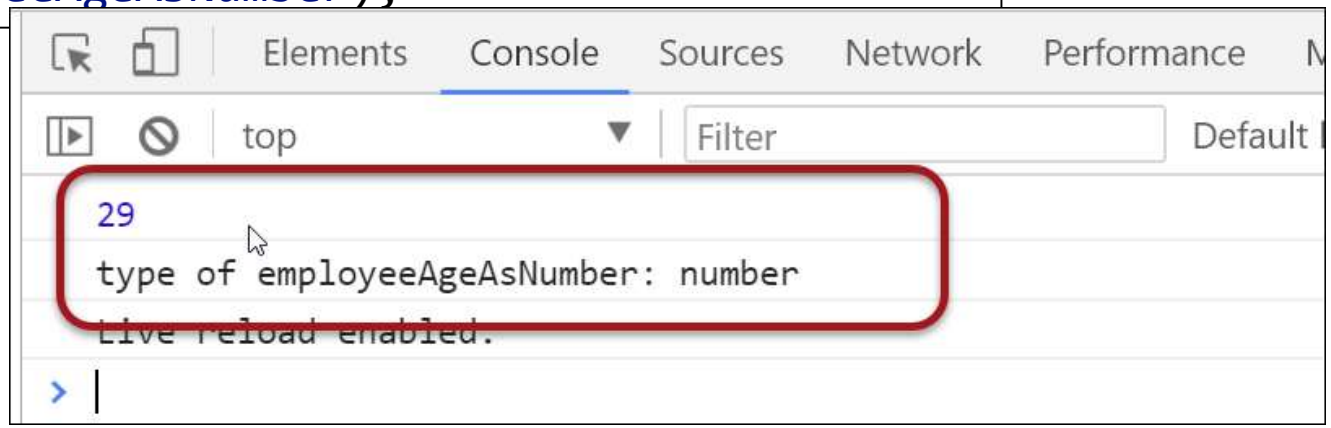
(but from an outside system (browser!), it might be passed in as a `string`).

So we write this helperfunction, using a type guard.

```
function getEmployeeAge(employee: Employee): number {
    // HERE, we implement the Type Guard
    if(typeof employee.age ==='number'){
        return employee.age; // simply return it. It’s already a number
    }
    return parseInt(employee.age); // convert to number, then return
}
```

Usage

```
const employeeAgeAsNumber =
    getEmployeeAge(new Employee('Dirk', '29'));

console.log(employeeAgeAsNumber);
console.log('type of employeeAgeAsNumber:',
    typeof employeeAgeAsNumber);
```



We've had conversion functions in JavaScript for ages, but by using a Type Guard, we are SURE our parameters are of a certain type.

`Instanceof` Type Guards

What does `instanceof` actually do? It compares the `prototype` of two objects.

If they are the same, the objects are apparently derived from the same instance.

```
// 1. simple example: what does instanceof actually do?
class Foo {
  something() {}
}

const bar = new Foo();
console.log(bar instanceof Foo); // true
```

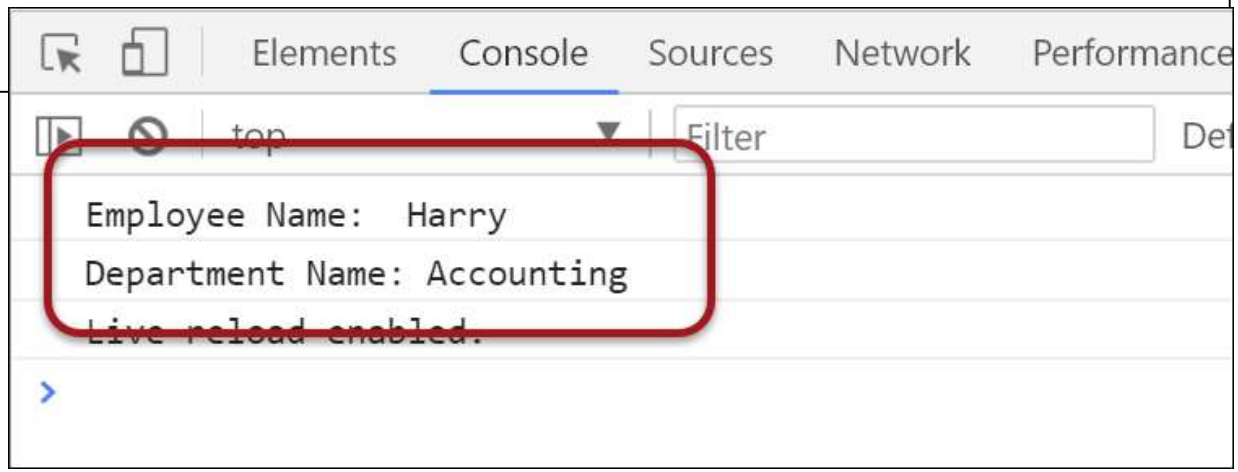
More real world example: get item name from a parameter

```
class Employee {
  constructor(public name: string, public age: string | number) {}
}

class DepartmentList {
  constructor(public title: string, public employees: Employee[]) {}
}
```

```
const employeeName = getItemName(new Employee('Harry', 52));
console.log('Employee Name: ', employeeName);

const departmentName = getItemName(
  new DepartmentList('Accounting', [
  new Employee('Astrid', 22),
  new Employee('Theo', 24)
  ])
);
console.log('Department Name:', departmentName);
```

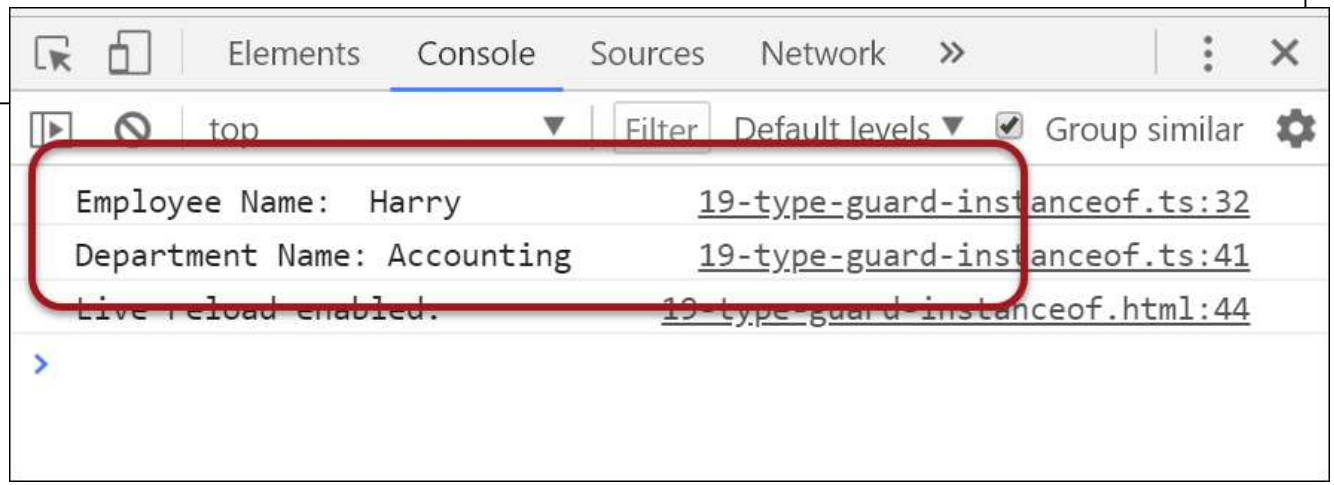
and the function `getItemName()`… (looks overly complicated with multiple castings)

```
function getItemName(item: Employee | DepartmentList): string {
  //ugly solution, cast each item
  if ((item as Employee).name) {
    // apparently we're dealing with an Employee
    return (item as Employee).name;
  }
  // we're dealing with a DepartmentList
  return (item as DepartmentList).title;
}
```



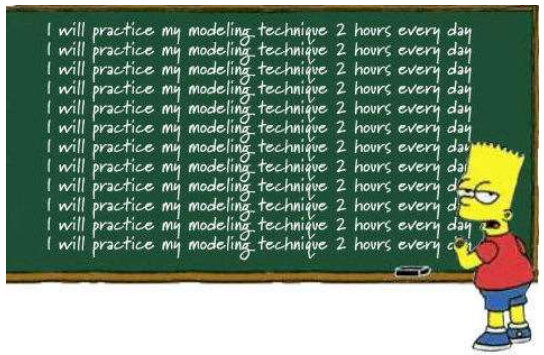
Cleaner solution – use `instanceof` type guard

```
function getItemName(item: Employee | DepartmentList): string {
  // Nice solution, use instanceof operator
  if (item instanceof Employee) {
    // We're dealing with an Employee
    return item.name;
  }
  // we're definitely dealing with a DepartmentList
  return item.title;
}
```



Workshop

- Create a function with a Type Guard that accepts a number or an array.
  - If an array is passed in, it returns the length of the array
    - `[ 1, 2, 3]; // returns 3,`
  - If a number is passed in, it returns the sum of the individual numbers.
    - `491; // returns 14 (=4+9+1)`
    - See `../18-type-guard-typeof.ts` as example
- Create two classes, create an `instanceof` Type Guard to return one of the properties.
  - See `../19-type-guard-instanceof.ts` as example



User Defined Type Guards / type predicate

- Create your own Type Guards by determining if a parameter is of a certain type: do so by creating a *Helper function*

```
function getItemName(item: Employee | DepartmentList): string {
  // Use custom function to determine if it is some type
  if (isEmployee(item)) {
    // We're dealing with an Employee
    return item.name;
  }
  // we're definitely dealing with a DepartmentList
  return item.title;
}

// Helper function: defer the type guard to a custom function
function isEmployee(item: any): item is Employee {
  return item instanceof Employee;
}
```

https://www.typescriptlang.org/docs/handbook/2/narrowing.html#using-type-predicates

By adding the `item is Employee` as the return type, we are casting the boolean result of the `instanceof` comparison back to the desired type!

https://www.typescriptlang.org/docs/handbook/2/narrowing.html#using-type-predicates

Example: `20-type-guard-user-defined.ts`

The ‘in’ type Guard

Checking if certain properties are available in a `<Type>`

TypeScript ‘in’ operator

- If we want to be more flexible than using `instanceof`, we can use the `in` operator.

“The `in` operator can be used to help TypeScript narrow the type of an object variable by its property name. It can be more useful than instanceof because it can be applied to any object structure.”

https://learntypescript.dev/07/l5-in-type-guard

Example

```
class Person {
    constructor(public name: string, public age: string | number) {
    }
}

class Employee {
    constructor(public company: string, public department: string) {
    }
}

// 3. The app creates a variable of a specific Type
const Arjan = new Employee('Google', 'Development')

// ---- somewhere else in the app ----
function printPerson(person: Person | Employee): string {
    // HERE, we implement the Type Guard using 'in'
    if ('company' in person) {
        return 'We are dealing with an Employee!';
    }
    return 'We are dealing with a plain Person';
}
```

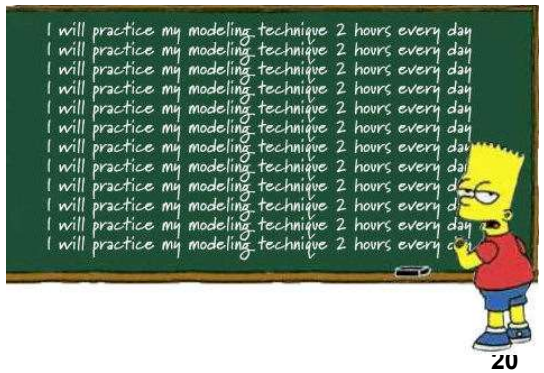
../18a-type-guard-in.ts

- Instead of using `strings` in the type guard we could also use `constants`. For instance:

```
const COMPANY = 'company'

// --- somewhere else ---
function printPerson(person: Person | Employee): string {
    // HERE, we implement the Type Guard using 'in'
    if (COMPANY in person) {
        return 'We are dealing with an Employee!';
    }
    return 'We are dealing with a plain Person';
}
```

Workshop

- Create two types (or: interfaces), a `Person` and an `Organization`.
- Create a union type `Contact`, which is a `Person` or an `Organization`
- Create a function `sayHello()` that takes a `contact` as its parameter.
  - The function should return `'Hello {firstName}'` if the passed in contact is a Person.
  - It should return `'Company name is {orgName}'` if the passed in contact is an Organization.
  - Use the `in` operator for type guards/checks
  - See `../18a-type-guard-in.ts` as example



The ‘assert’ type Guard

Assert a specific type, otherwise throw error

We assert a certain type

```
// 1. helper function that  asserts some input is of type 'string'
function assertIsString(val: any): asserts val is string{
    if(typeof val !== 'string'){
        throw new AssertionError('This is not a string!')
    }
}

function sayHello(input : any){
    assertIsString(input);
    // we're definitely dealing with a string.
    // You can use the asserts guard to be very explicit.
    return `Hello ${input}!`
}

console.log(sayHello('David')); // OK, since we're dealing with a string
```

../20a-type-guards-assert.ts

Workshop

- Create a types (or: interface) `Person`
- Create a constant, based on that type
- Create a helperfunction that accepts an `any` type and checks if the passed in argument is of type Person.
  - If not, throw an error
- Create a function that accepts an `any` type and check if the passed in argument is a `Person`.
  - If yes, print the `person` to the console
  - See `../20a-type-guard-assert.ts` as example