Type Guard

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

typeof

TypeScript is aware of the usage of the JavaScript `instanceof` and `typeof` operators. If you use these in a conditional block, TypeScript will understand the type of the variable to be different within that conditional block. Here is a quick example where TypeScript realizes that a particular function does not exist on `string` and points out what was probably a user typo:

```
function doSomething(x: number | string) {
    if (typeof x === 'string') { // Within the block TypeScript knows that `x` must be a string
        console.log(x.subtr(1)); // Error, 'subtr' does not exist on `string`
        console.log(x.substr(1)); // OK
    }
    x.substr(1); // Error: There is no guarantee that `x` is a `string`
}
```

instanceof

Here is an example with a class and `instanceof`:

```
class Foo {
    foo = 123;
    common = '123';
}

class Bar {
    bar = 123;
    common = '123';
}

function doStuff(arg: Foo | Bar) {
    if (arg instanceof Foo) {
        console.log(arg.foo); // OK
        console.log(arg.bar); // Error!
    }
    if (arg instanceof Bar) {
        console.log(arg.foo); // Error!
        console.log(arg.bar); // OK
    }

    console.log(arg.common); // OK
    console.log(arg.foo); // Error!
    console.log(arg.bar); // Error!
}

doStuff(new Foo());
doStuff(new Bar());
```

TypeScript even understands `else` so when an `if` narrows out one type it knows that within the else *it's definitely not that type*. Here is an example:

```
class Foo {
    foo = 123;
}

class Bar {
    bar = 123;
}

function doStuff(arg: Foo | Bar) {
    if (arg instanceof Foo) {
        console.log(arg.foo); // OK
        console.log(arg.bar); // Error!
    }
    else {  // MUST BE Bar!
        console.log(arg.foo); // Error!
        console.log(arg.bar); // OK
    }
}

doStuff(new Foo());
doStuff(new Bar());
```

in

The `in` operator does a safe check for the existance of a property on an object and can be used as a type guard. E.g.

```
interface A {
  x: number;
}
interface B {
  y: string;
}

function doStuff(q: A | B) {
  if ('x' in q) {
    // q: A
  }
  else {
    // q: B
  }
}
```

Literal Type Guard

When you have literal types in a union you can check them to discriminate e.g.

```
type Foo = {
  kind: 'foo', // Literal type
  foo: number
}
type Bar = {
  kind: 'bar', // Literal type
  bar: number
}

function doStuff(arg: Foo | Bar) {
    if (arg.kind === 'foo') {
        console.log(arg.foo); // OK
        console.log(arg.bar); // Error!
    }
    else {  // MUST BE Bar!
        console.log(arg.foo); // Error!
        console.log(arg.bar); // OK
    }
}
```

User Defined Type Guards

JavaScript doesn't have very rich runtime introspection support built in. When you are using just plain JavaScript Objects (using structural typing to your advantage), you do not even have access to `instanceof` or `typeof`. For these cases you can create *User Defined Type Guard functions*. These are just functions that return `someArgumentName is SomeType`. Here is an example:

```
/**
 * Just some interfaces
 */
interface Foo {
    foo: number;
    common: string;
}

interface Bar {
    bar: number;
    common: string;
}

/**
 * User Defined Type Guard!
 */
function isFoo(arg: any): arg is Foo {
    return arg.foo !== undefined;
}

/**
 * Sample usage of the User Defined Type Guard
 */
function doStuff(arg: Foo | Bar) {
    if (isFoo(arg)) {
        console.log(arg.foo); // OK
        console.log(arg.bar); // Error!
    }
    else {
        console.log(arg.foo); // Error!
        console.log(arg.bar); // OK
    }
}

doStuff({ foo: 123, common: '123' });
doStuff({ bar: 123, common: '123' });
```