Type Checking with instanceof

This lesson delves further into type checking with instanceof, and how TypeScript uses it to compare types.

WE'LL COVER THE FOLLOWING

`instanceof` with classes

`Instanceof` also comes from JavaScript and executes at runtime. Nevertheless, TypeScript can handle the test operator that is looking at the prototype chain to figure out the type of object. Like `typeof`, if the comparison with `instanceof` is positive, the type is narrowed down to the checked type. `Instanceof` cannot be used for interface checks and is limited to a class comparison since it relies on the constructor function.

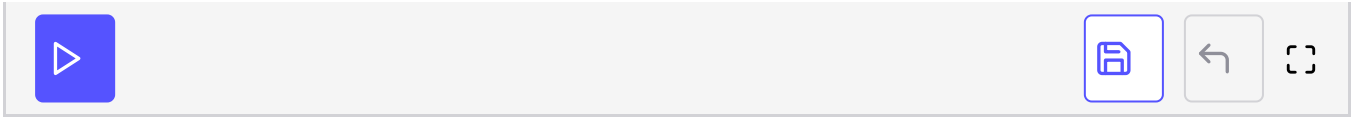
In the following example shows the `C2` inherits `C1`. At **line 3**, an instance of `C2` is created but when checked to see if it is `C1` is is `true`. **Line** 7 also returns `true`. This peculiarity is since **TypeScript 2.1** in regard to inheritance and `instanceof`.

```
class C1 { m1!: string }
class C2 extends C1{ m2!:number }
const c2 = new C2();
if (c2 instanceof C1) {
  console.log("c2 is an instance of C1");
}
if (c2 instanceof C2) {
  console.log("c2 is an instance of C2");
}
```

The previous example makes it impossible to distinguish between the two

classes with `instanceof`. In the following example, both classes are distinguished. The reason is that they are in two different prototype chains.

```
class C1 { }
class C2 { }
const c1 = new C1();
if (c1 instanceof C1) {
  console.log("c1 is an instance of C1");
}
if (c1 instanceof C2) {
  console.log("c1 is an instance of C1");
}
```

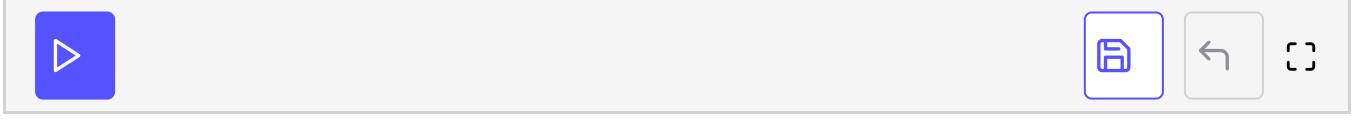


Even if the two classes have the same structure, they are distinct with `instanceof`.

`instanceof` with error, array and map

However, if any of the prototype chains extend `Error`, `Array` or `Map`, then each constructor must call the `setPrototypeOf` in each constructor.

```
class C100 extends Error {
  constructor() {
    super();
    Object.setPrototypeOf(this, C100.prototype);
  }
}
class C200 extends C100 {
  constructor() { super(); Object.setPrototypeOf(this, C200.prototype); }
}
const c100 = new C200();
if (c100 instanceof C100) {
  console.log("c100 is an instance of C100");
}
if (c100 instanceof C200) {
  console.log("c100 is an instance of C200");
} if (c100 instanceof Error) {
  console.log("c100 is an instance of Error");
}
```



Before **version 2.7**, `instanceOf` operator relied on the structure of the object. In **version 2.7** and after, the operator uses the inheritance chain.

The following example shows the difference. The `ChildB` and `ChildC` have the

same structure as defined at **line 2** and **line 3**.

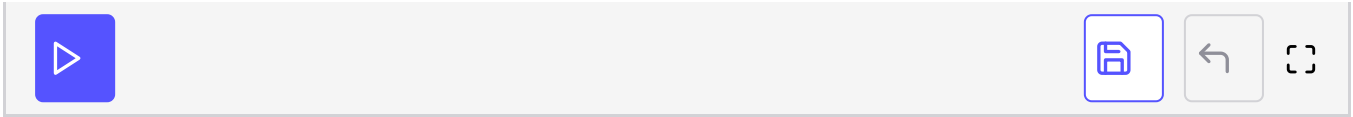
In versions prior to **2.7**, `instanceof` compares by the structure and hence the first conditional statement accepts `ChildB` and `ChildC`. It means it would have gone inside the condition at **line 7**. The `else` condition is never reachable because of the two conditions (B + C and D) before this one covers all cases.

In version **2.7**, the first conditional block only lets `ChildB`, the second block `ChildC` and all others `ChildD`. Hence, the following code execution will get in each condition's statements.

```
class BaseClassA { }
class ChildB extends BaseClassA { }
class ChildC extends BaseClassA { }
class ChildD extends BaseClassA { c: string = "c value" }

function compareWithInstanceOf(x: ChildB | ChildC | ChildD) {
  if (x instanceof ChildB) {
    console.log("Found an instance of B");
    x;  // B (V2.6: B | C)
  } else if (x instanceof ChildD) {
    console.log("Found an instance of D");
    x;  // D
  } else {
    console.log("Found an instance of C");
    x;  // C (V2.6 never)
  }
}

compareWithInstanceOf(new ChildB);
compareWithInstanceOf(new ChildC);
compareWithInstanceOf(new ChildD);
```



The output of the previous code is exactly in the order of the calls at **line 19-21**. The reason is the code is running on a **version above 2.6**. The `instanceof` checks that the left part derives from the class compared and with multiple conditions it is possible to narrow precisely.