The Curly Braces Object

This lesson explains the creation of objects by using curly braces.

WE'LL COVER THE FOLLOWING

Object literal advantage

TypeScript can create an object using the curly braces – it is an object literal like in JavaScript. The limitation is that you must define every member right at initialization time.

The advantage is that it's a quick way to organize data. It's also a natural way to organize data coming from a JSON Payload. For example, executing a request to receive a payload will provide you with a literal object.

TypeScript is a structural language, and hence does **not** need to have a name. This works great for cases where you do not need to map all the data to a type. With a structural type, you can cast the data and have the structure mapped for you without having to instantiate anything.

How to specify type to a curly object

When not working with data coming from a third party or across the wire, you can use a variable and specify the type using a colon and then use the curly braces to set the value. TypeScript will catch (at compilation time) that something is missing if the type changes.

```
let x: { x: number; y: string } = { x: 1, y: "2" }; // Below code is similar, but reusable wi
let x: { x: number; y: string } = { x: 1, y: "2" }; // Below code is similar, but reusable wi
interface MyTypeWithTwoMembers {
    x: number;

    y: string;
}
let x2: MyTypeWithTwoMembers = { x: 1, y: "2" };
```

Structural language

However, this is not the case if the type is not defined, since it will rely on the new structure and might continue to fit in subsequent code. An important detail is that instead of defining a variable with a type, using the colon, just relying on the equality to set values, and then casting to force the type, it will set the variable values even if not complete and will allow you to use the variable with the type. The following code, at **line 1** defines an object with two properties. If you move your cursor above the variable, you can see the type of the object that has a `string` and a `number`.

```
let stronglyTypedObject = { name: "John", age: 98 };
console.log(stronglyTypedObject);
```

The dangers of casting

Casting is dangerous since you might end up having a typed variable that is not fully structured, thus getting a runtime error.

The following example is built on the previous one. The example shows a member called `x` of type number. The value is set to `1` and (if we do not cast) would be of an anonymous type that accepts only the member `x` of type `number`. However, since we cast to an interface, the actual type of the variable is the same as the interface.

However, the member `y` is missing which would result in an unexpected result at execution time since `y` was not defined to be of type string or undefined but accepting strings.

```
let myObjectTypedWithCurlyType3 = { x: 1 } as MyTypeWithTwoMembers;
```

It’s important to note that casting curly braces to should not be

It's important to note that casting curly braces to `any` should not be considered a viable strategy, since it opens the door to giving members

unexpected values. This results in the loss of the types. Casting to an interface or type should be kept to a minimum in your code. One valid use case is when receiving a payload from an Ajax call, which will always come as the type `any`. Casting is allowed since the API should return the response in a format that is well known and matches your TypeScript-defined interfaces/types. This approach doesn't guarantee validation if Ajax's response contract changes. In the case of a change, the code will still compile, but will fail at runtime when the expected value is undefined. Casting will be discussed later in this course.

On a final note, objects created with a curly bracket have access to all members of the Object type (capital O). Further details are described shortly in the "Lowercase vs Uppercase Object" lesson.