Scope

This lesson will explain how the scope can affect the lifetime of an identifier.

Global Scope

Throughout this section, we've been creating and manipulating data in the outermost layer of our ReasonML program. This layer is known as the **global scope.**

Variables created in the *global scope* are accessible everywhere in the program globally.

Local Scope

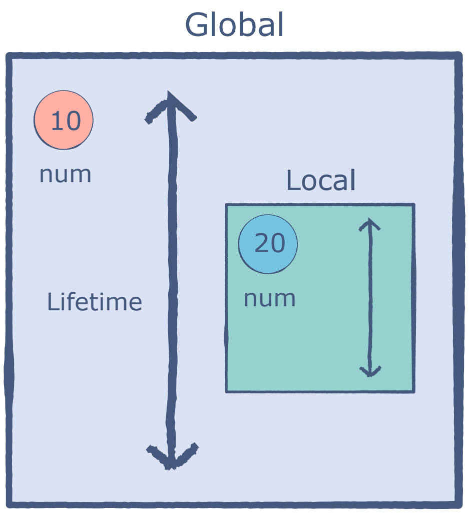
As we'll see later in the course, several operations take us into an inner layer, which is also known as a **local scope**. This is a contained environment. The variables and data generated in a local scope are not accessible from outside.

As soon as a local scope ends, all its data is removed from memory and cannot be recovered. In this sense, a local scope determines the lifetime of the local identifiers inside it.

A local scope is enclosed within the `{}` brackets. Let's take a look at an example:

```
let num = 10;
/* A local scope */
{
  Js.log(num); /* 10 */
  let num = 20;
  Js.log(num); /* 20 */
  Js.log(num); /* 20 */
}
Js.log(num); /* 10 */
```

We can observe in the code above that the global definition of `num` is accessible to the local scope in **line 4**. However, the local definition is not available after the scope ends.



This following code would produce an error:

```
{
  let num = 10;
};
Js.log(num);
```

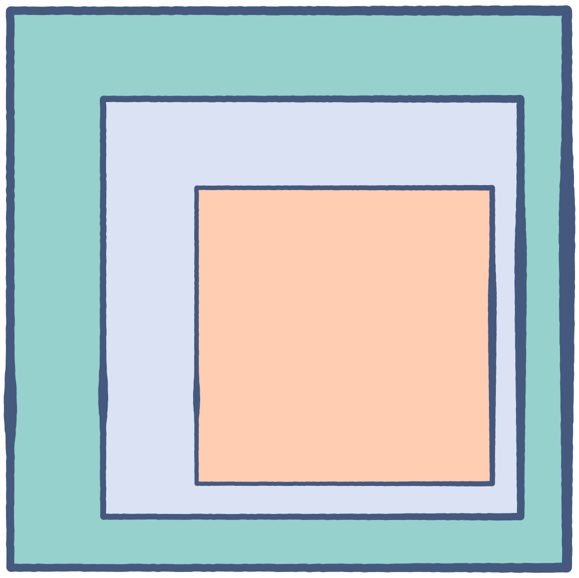
Why wouldn't it work?

This is because the global scope does not know what `num` is. The variable was destroyed as soon as the local scope ended.

Nested Scopes

Reason allows us to create scopes within scopes, each with their own variables and operations. This is essential for many programming essential for many programming concepts such as **functions** and **conditional statements**. All of these concepts will appear in due time.

Scopes within scopes



Now that we’re done with basic `let` bindings, we can move on to the second type of identifier. See you in the next lesson.