NetsBlox Lesson: Introduction to Lists
Movie App

Variables can store values, like a numbers or text. To create a variable, go to the Variables tab and click the small grey button labeled “Make a variable.” In the window that pops up, you need to provide a name. A meaningful one is preferred that tells us something about the value the variable will store. There is an option whether we want the variable to be accessible only in the current sprite or across the entire program we are working on. For this lesson, we can just leave the default.

Once we click OK, the variable shows up in the Variable tab (along with the “Delete a variable” button).

The checkbox is used to select whether we want to the see the variable in the Stage or not. Once we have a variable, we can set it to store a value:

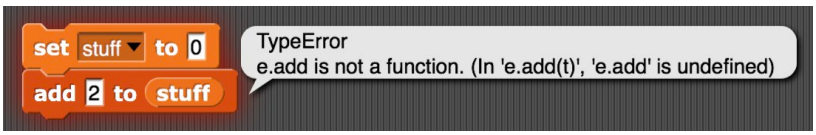
However, a variable can also contain a list of values (or items, as we typically say). That is a very powerful feature that is used pretty widely in programming. The variables tab has several blocks related to lists. However, we do not need them all just yet. Here are the most important ones:

One important thing to remember is that we have to specify if we want a variable to contain a list. We cannot add an item to a variable if we did not make it a list first. This is what to do:

and the variable `stuff` will look like this after executing the block:

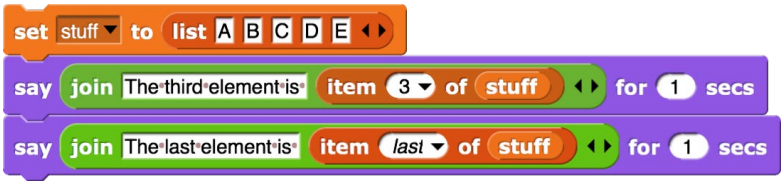
The arrow keys on the `list` block allows us to add or remove items. If we do not want to add items initially, we still need to make the variable a list. Simply eliminate all item placeholders from the `list` block by clicking the left arrow. This is what we should have:

If we forget to make the variable a list and try to add an element to it, we get a somewhat cryptic error message:



Notice the red highlight around the blocks to. If you see this, remember that the variable needs to be a list before we can add an item to it.

As you see above, to add an element to the end of a list, simply use the `add` block. To access individual elements, we have to use the index concept. You can simply think of lists as if their elements are numbered. At the beginning of the list is the first element or item #1, then comes the second or item #2, etc. Then we can simply access elements with the `item` block:



You should see C and then E after a second. In the `item` block, we can type in an index, drag in a variable or use the pull-down menu for things like “last” or “random.”

Notice how we used the green `join` block (Operators tab) to combine pieces together to form some text and then displayed them with the `say` block (Looks tab).

To see how long a list is, that is, how many elements it has, use the `length` block. The other list blocks are fairly self-explanatory, so feel free to try them out on your own.

Our first exercise is to create a list of 10 random numbers:

Not much explanation is needed, right? And this simple program illustrates the power of loops and lists: by simply changing the 10 to a thousand, we can create a list of one thousand random numbers with just four blocks of code.

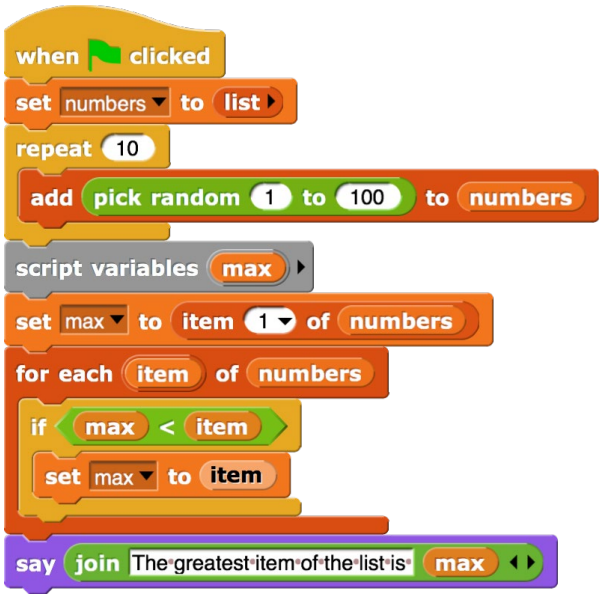
As a more complicated task, let's write a program that finds the largest item in a list. Now we need to start to think first and come up with an algorithm before we start coding. Think whether you can figure it out, before continuing…

Here is the basic idea: go through the list one by one and compare the current item to the max we found so far. If it is larger than the current max, we replace the current max with the current element, otherwise, we do nothing. This works quite well, but there is one more important consideration: what should be the value of the currently found maximum when we start?

If we know that all the numbers in the list are positive, for example, we can set that value to zero. That ensures that we will find a greater value at least once and hence, find the correct max. But what if there is no lower limit? How can we make sure that we will pick an element from the list when we do the comparison? For example, if the list can have negative values too, zero would be a bad choice. Why? What if all the elements are negative, then zero is greater than all of them, so our algorithm will find the wrong value, zero.

The solution is to pick one item of the list as the initial max. Typically, you want to use the first item, but it does not really matter which one. If you do this, then there are two cases: 1) either this item is the max and then we'll find the correct value since the algorithm will never change it or 2) there is at least one greater element and then our algorithm will find it.

Now, we are ready to turn our algorithm into code:



The `script variables` grey block allows us to define temporary variables that will be only visible in the script they were defined at. The `for each` loop simply iterates through the elements of a list: the variable `item` will take on each item one by one from first to last.

Feel free to experiment with this program by creating a list of thousands of random numbers and see how our program finds the maximum quickly. If you do that, however, remember to turn on Turbo Mode (settings/cogwheel menu). Otherwise, the program may take a while to complete.

Now, we are ready to put lists into practical use by creating a Movie App.

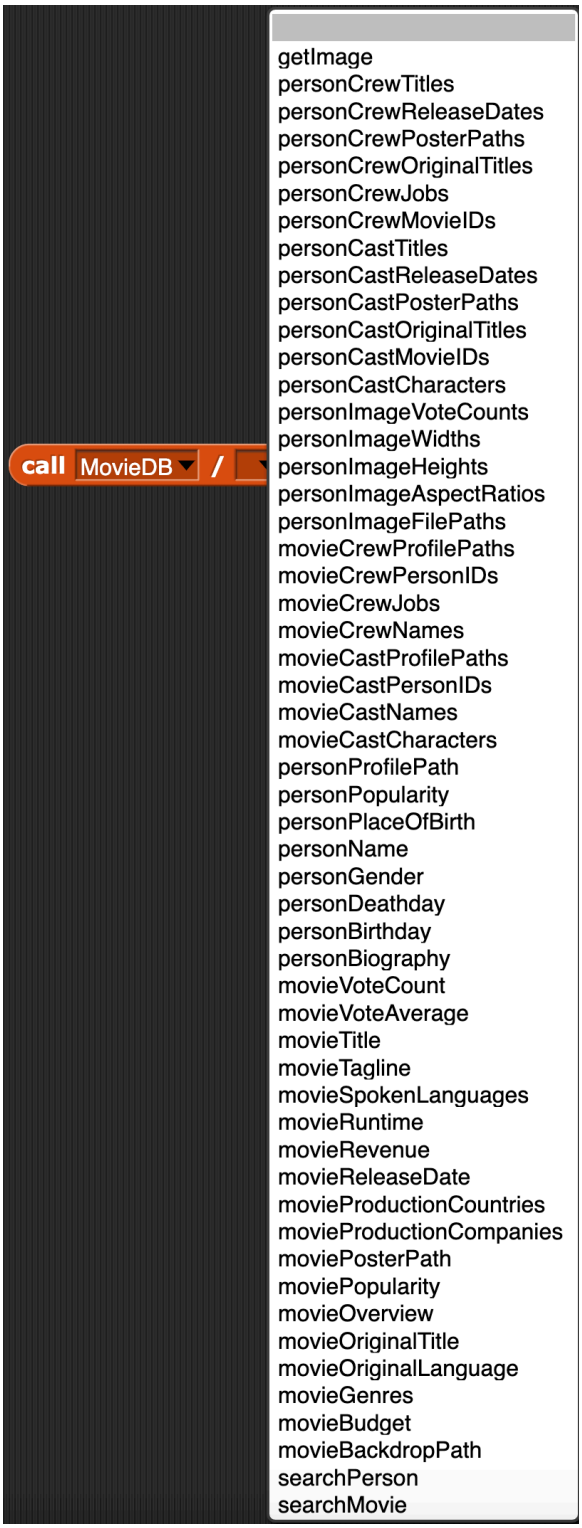
The MovieDB service

One of the core networking/distributed computing concepts of NetsBlox is Remote Procedure Calls (RPC). An RPC is similar to a custom block: it is some code that can be executed using some input parameters. An RPC is a typically reporter, that is, it reports (returns) some result. The major difference is that an RPC runs remotely, specifically, on the NetsBlox server in the cloud.

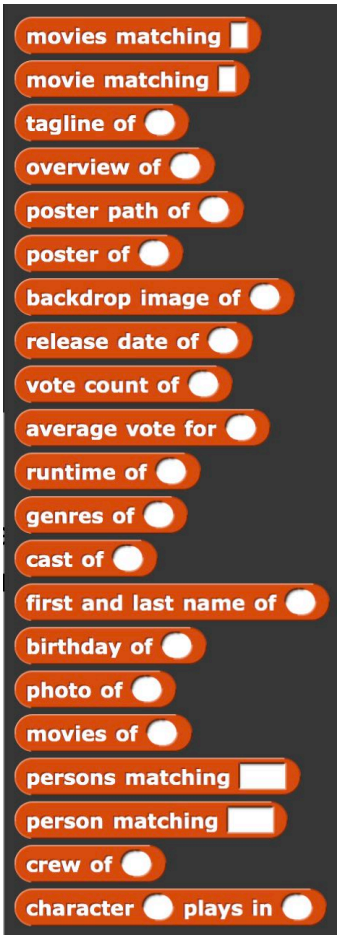
NetsBlox uses RPCs to provide access to interesting data sources and services already available on the web. Related RPCs are group together into Services. NetsBlox has many of them including Google Maps, Weather, Earthquake, Geolocation, etc.

To use RPCs is very easy. Simply go to the *Network tab* on the palette and use the `call` block. It has two pull down menus. The first selects the Service from the list of all available ones:

The MovieDB service provides functionality similar to that of IMDb. It has quite a few RPCs:



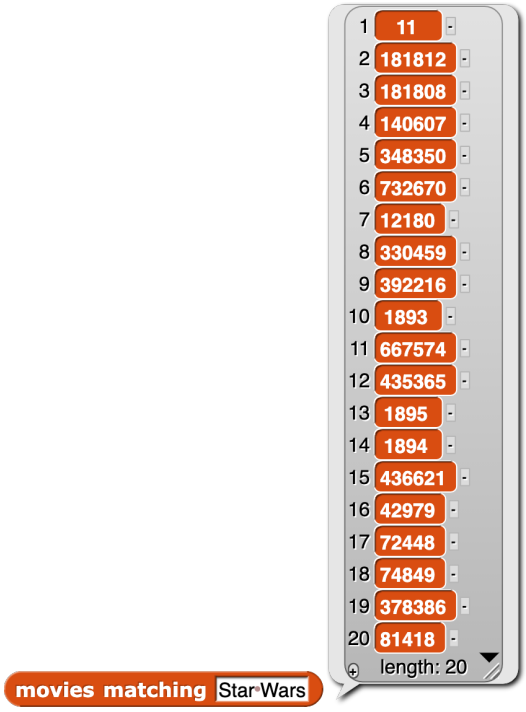
On smaller screens, you may not be able to see all of them. Fortunately, most services come with several predefined custom blocks. In the File Menu select Services and then from the list that pops up click Movie database. After a few seconds, the custom tab will have several new blocks:



Most of these are simply wrappers around RPC calls. In other words, all they do is call and RPC and return the result. For example, if you right click on the `movies matching` block and select Edit, this is what you'll see:

It is up to you whether you prefer these custom blocks or calling the RPCs directly.

Let's see what happens when we actually use one of these:



What is this list of numbers? Each movie in the database is associated with a unique ID and that is what you see here. Just like phone numbers, social security numbers or email addresses, we need a way to identify things that is guaranteed to be unique. We cannot use the title of the movie since more than one may have the same title. Hence, the need to work with IDs.

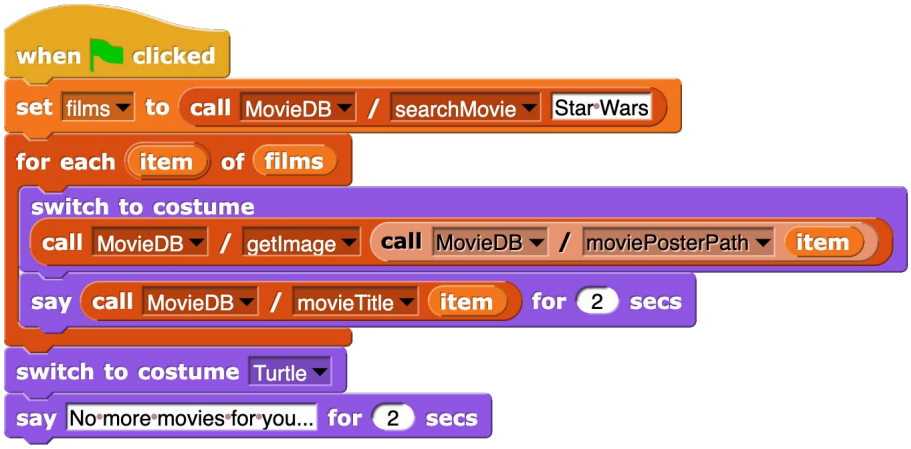
The RPC returns the IDs of all the movies whose title contains the word Star Wars (to be more precise, the first 20 if there are more than 20).

To make use of it, let's see what the title of the first movie on that list:

First, we call the `movies matching` custom block (that, in turn, calls the appropriate RPC), then we take a random item of the reported list and then use that movie ID as an input to the `movieTitle` RPC which returns the title, *Star Wars: The Rise of Skywalker*.

The Movie App

Now we are ready to write an actual program about movies: Let's cycle threw all the Star Wars movies and show their posters:



First, we create a variable called `films` and assign all the movies that match Star Wars to it. Then we use the `for each` loop that is the best option to visit all items in a list. There are two new RPCs we use here: the `moviePosterPath` will return a URL that identifies the image file in the database. So, we use it as an input to the `getImage` RPC to get the actual image and switch the costume to it.

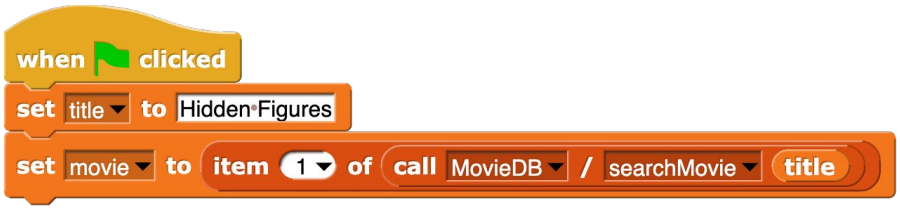
Since some movies do not have posters, we display the movie titles also. Here is a link to the program (with some minor differences; see whether you can spot them…):

https://editor.netsblox.org/?action=present&Username=ledeczi&ProjectName=MoviePosters

Instead of the poster, let's show a photo of the leading cast member of a movie!

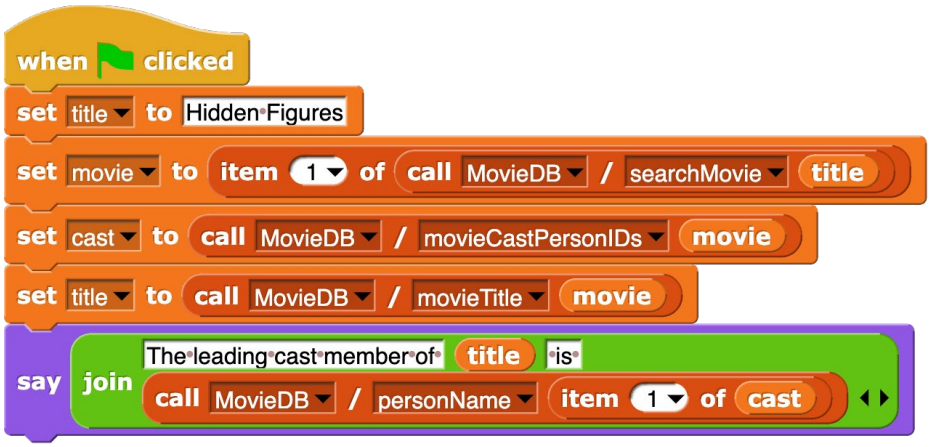
Let's start from scratch by creating a new project using the file menu. Do not forget to save your work first. You may event want to export it to have a local copy. Export creates an xml file in the downloads folder. To import such a file, you can use the File menu Import command or simply drag and drop the file on the canvas.

Once we have a new project, let's create three variables: `title`, `movie` and `cast`. Let's set the title to our favorite movie, for example, Hidden Figures by simply typing in the name. Then let's use the `searchMovie` RPC to get a list of matching movies and pick the first one using the `item` block:



Once we have the ID of the movie, we can get its cast with the `moviecastPersonIDs` RPC (a mouthful, isn't it?). Note that this returns a list of IDs, but these IDs now refer to people: actresses and actors. (The database also contains directors, producers, etc.) Form the cast list, we can simply pick the first one because this list is ordered by the importance of the role.

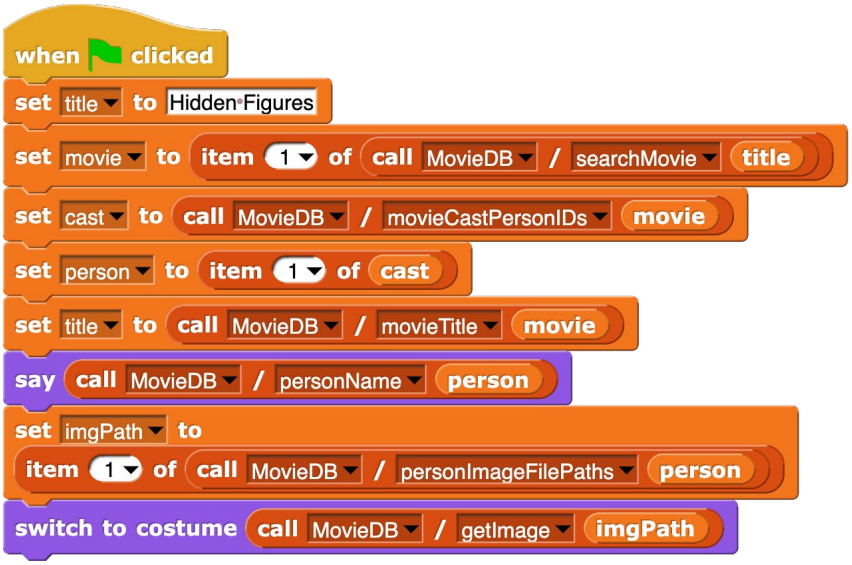
There is one more detail to consider: the `title` variable currently holds the text we typed in ourselves. If it does not exactly match the title, we may still get a lot of movies with similar titles. Consider the previous example where we got 20 different movies for the title "Star Wars." Now that we have the ID of the movie, we can simply request its exact title using the `movieTitle` RPC. Finally, we can display the information we currently have with a `say` block and the `personName` RPC:

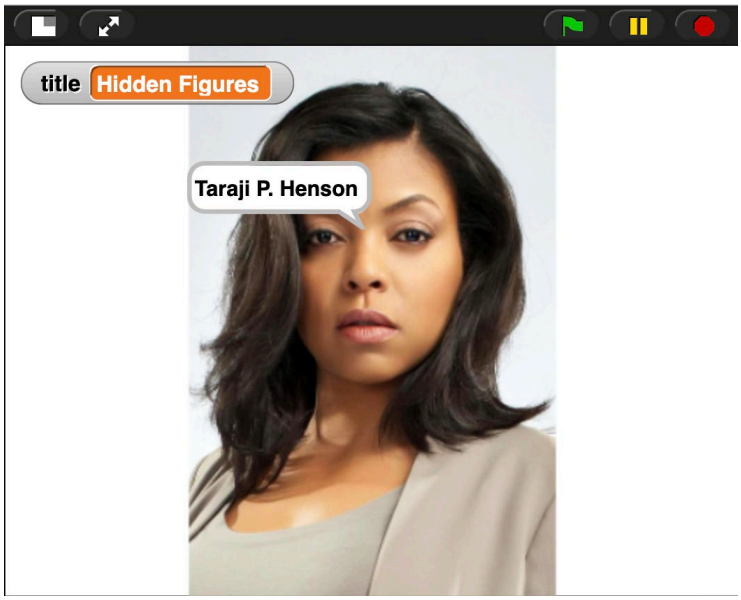


***Practice**: Use a loop to display the leading cast member of every movie in the list.*

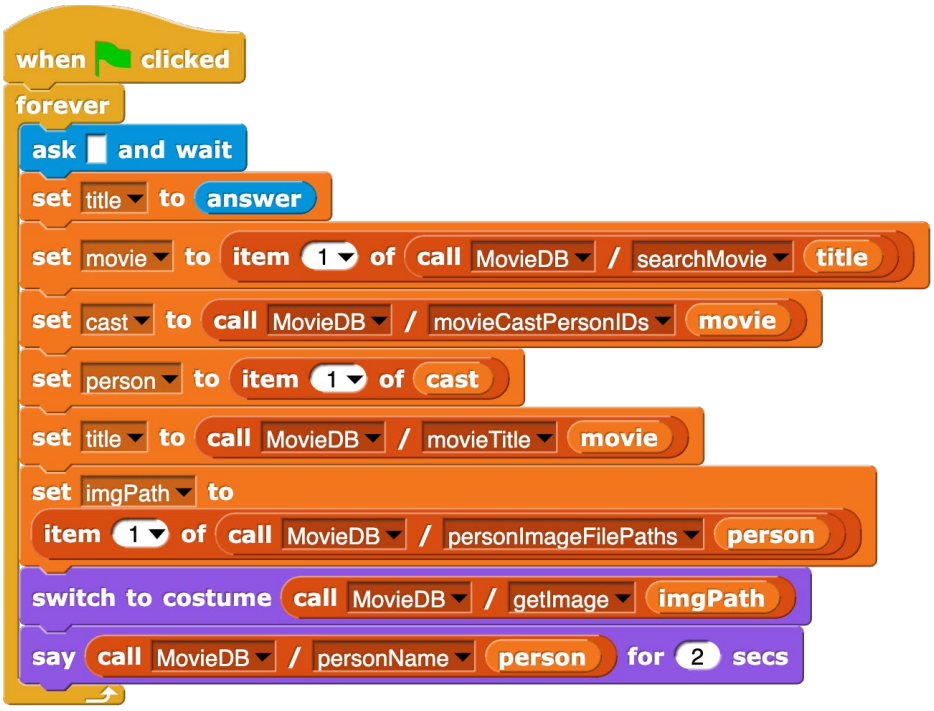
Let's display a photo of the leading cast member next! Let's add two new variables: `person` and `imgPath`. The variable `person` will hold the ID of the person and the `imgPath` will store the file location of the photo. We'll use the `personImageFilePaths` RPC to get the list of files of all the photos of the given person and just pick the first one. Then we use the same `getImage` RPC we used with the posters.

We have also shortened the text we display with the `say` block not to cover up too much of the photo.





Next, we should make the program ask the user for a movie title and show the leading cast member of it. Under the Sensing tab, there is the `ask` and `answer` blocks. We can simply use them to get the user type a new title. Also, we can put the entire script into a `forever` loop. We do not include a prompt, such as, “Enter a title” because that would cover up the photo. We also change the `say` block to `say for _ seconds` to show the name for a while, but then remove it to be able to see the entire picture. Here is the code:

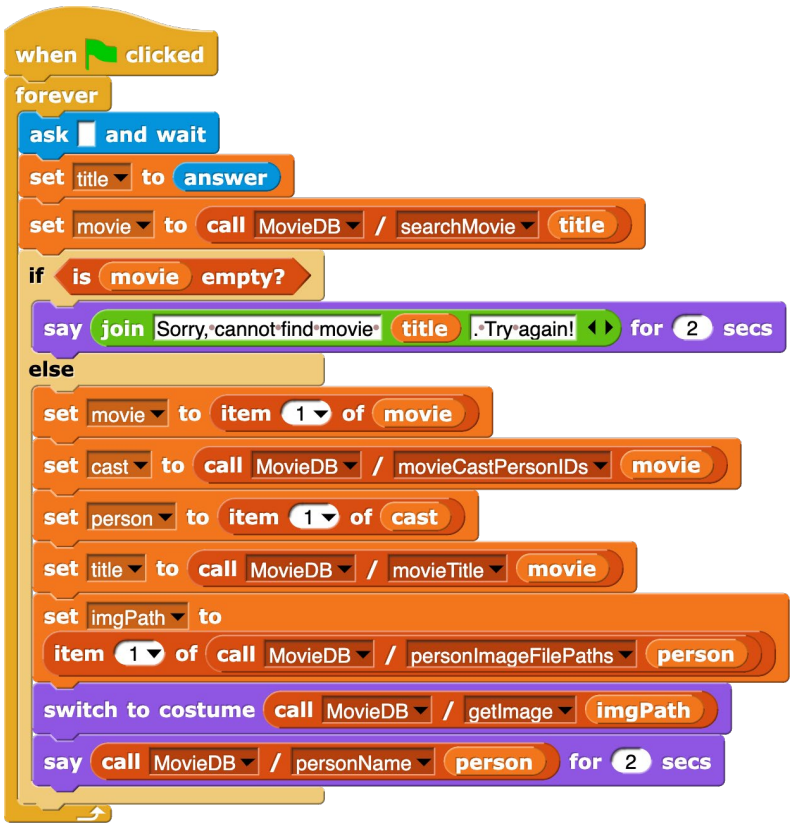


Very nice! We have a nice interactive program that gets its data from the internet and it has access to thousands and thousands of movies. Are we done? Not quite.

Try typing a crazy movie title like "fiuq bfyq kbsvb io fs puheqr foi!" What happened? We got an error message from the block that is trying to set the `cast` variable saying something like "Error expecting list but getting text." The script also got this nice red halo effect and the program quit. Why? Our search for a movie returned an empty list since we did not find any matching movies. Then we are trying to use its first element to set the `movie` variable, which is nothing. Then we try to get the cast of nothing, etc. You get the picture. The moral of the story is that before we try to get an element of a list that may be empty, we need to make sure it is not.

Our good friend, the `if` block, or more precisely an `if-else` block, comes to the rescue. We simply check whether the list is empty or not and only proceed if it is not. We can display our own error message, if it is empty.

First, we set the movie variable to the entire list returned by the `searchMovie` RPC. We check whether it is empty. If it is, we tell the user. If it is not, we set the `movie` variable to the first element of the list that is stored in the `movie` variable currently. But once we execute that block, it stores only a single ID, the first element of the original list. (We could also use two separate variables: one for the list of movies and one for the one movie we are interested in.)



Much better, but we are still not quite done. There are two more places where we get the first item of a list without checking. What if a movie does not have a cast list specified? What if a given person does not have a photo? Currently, we get another error message and the program quits. We leave this as an exercise for the reader.

However, here is a link to the finished project:

https://editor.netsblox.org/?action=present&Username=ledeczi&ProjectName=VDN-CastShow

***Extension #1**: Display all the photos the database has of the cast member! Cycle through them in a loop. Hint: save the photos in a list! Calling the RPC `getImage` (or any other RPC) all the time is wasteful.*

***Extension #2**: Display the three leading cast members, not just one. Use three sprites, positioned next to each other and sized so that they fit the stage. Try not to repeat too much code across the sprites! Hint: use a custom block…*

Here is the final project, if you want to take a peek:

https://editor.netsblox.org/?action=present&Username=ledeczi&ProjectName=CastShow&