Predictive & Live Debugging (/topics/153-predictive-live-debugging/)

Predictive Debugging

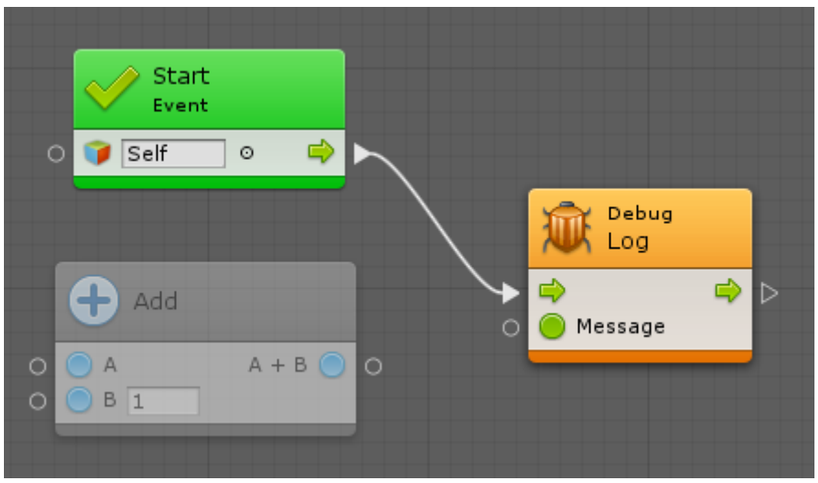
Bolt will attempt to predict nodes that may cause an error before you even enter play mode.

When a node is not properly configured or may cause an error, it will be colored **yellow**.

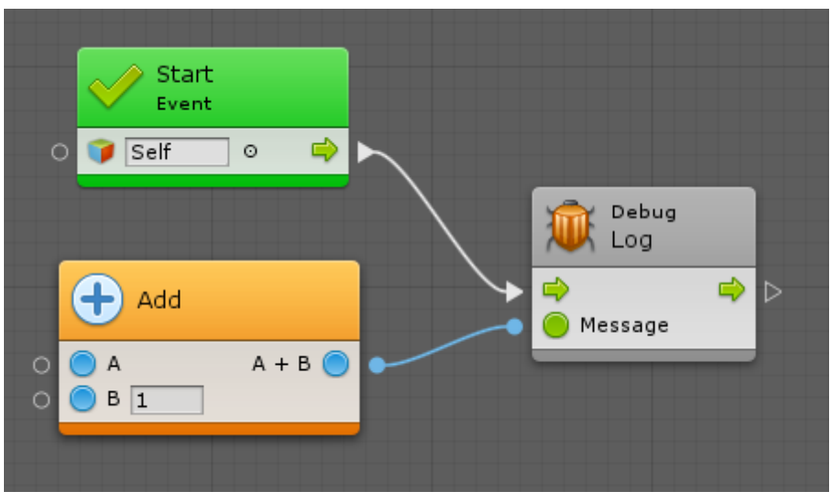
When a node is certain to cause an error, it will be colored **orange**.

In both cases, you should examine the node and make the required changes until it turns back to its normal color.

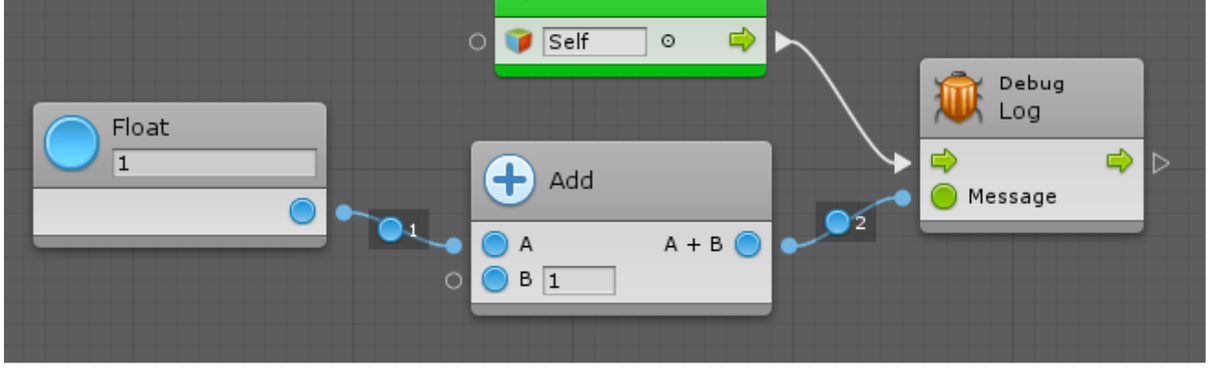
Let's look at a simple example. Here, the `Log` node is colored orange because it's missing the `Message` that it should output to the console:



If you connect the result of `A + B` to `Message`, the `Log` node will go back to normal. However, the `Add` node will turn orange, because it's missing its first operand, `A`.



If we properly provide values for both operands, everything will go back to normal.



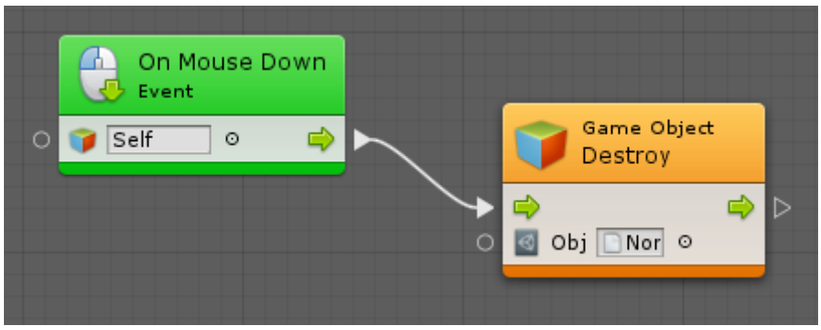
Note that we don't have to connect the B input port, because it has a default inline value.

Null References

Null reference exceptions are very common. They happen when a parameter expects a value, but you give it "nothing", or in scripting lingo, "null".

Bolt will attempt to predict those if the Predict Potential Null References option is checked in Preferences > Bolt > Flow Graphs .

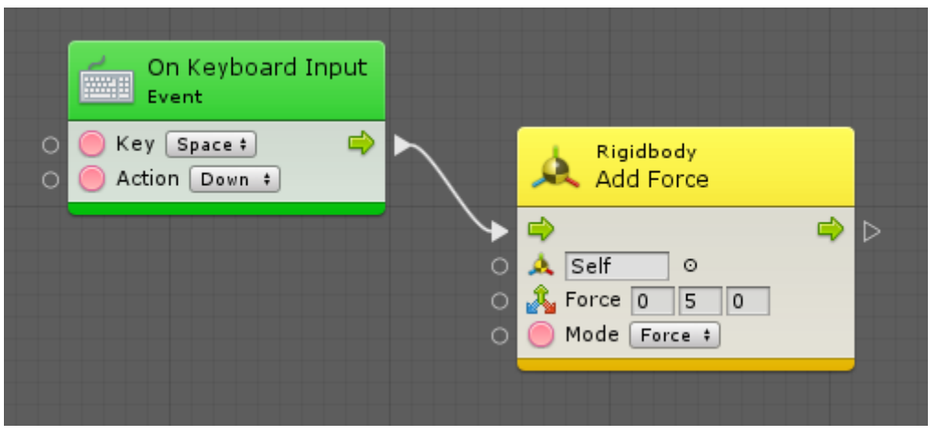
For example, even though the Destroy unit here has an inline value, since it is set to "None" (null), it is colored orange:

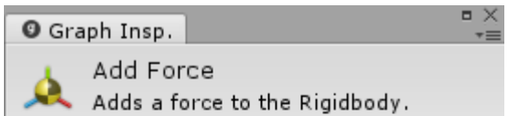


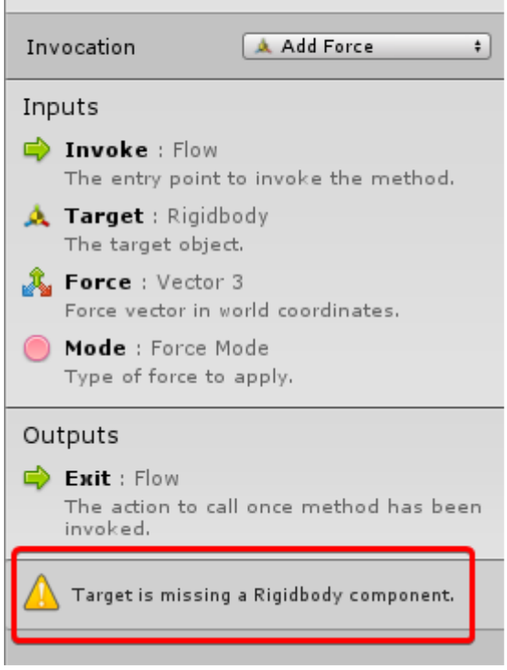
However, there are some rarer nodes that allow for null parameters. Unfortunately, because there is no way to know that from codebase analysis, Bolt will color them orange as a false positive. If this is a recurring issue for you, you can turn off Predict Potential Null References .

Missing Components

When you use nodes that require components and pass a game object or component that does not have the specified component, the node will be colored yellow as a warning. For example, here, even though we are providing default values for each value input of the Add Force unit, Bolt detects that the owner game object does not have a rigidbody and warns us:







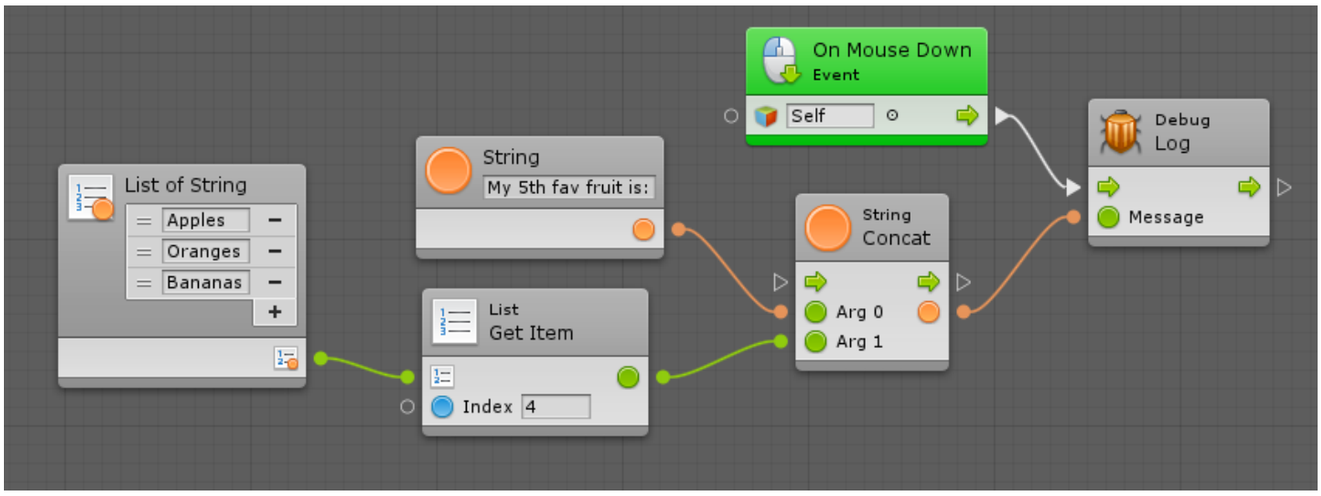
Bolt will not color it orange because it is possible to add components to game objects at runtime, so the node is not *guaranteed* to cause a crash if you add the required component before calling it. If this use case happens often in your project, you can disable Predict Potential Missing Components debugging from Preferences > Bolt > Flow Graphs .

Live Debugging

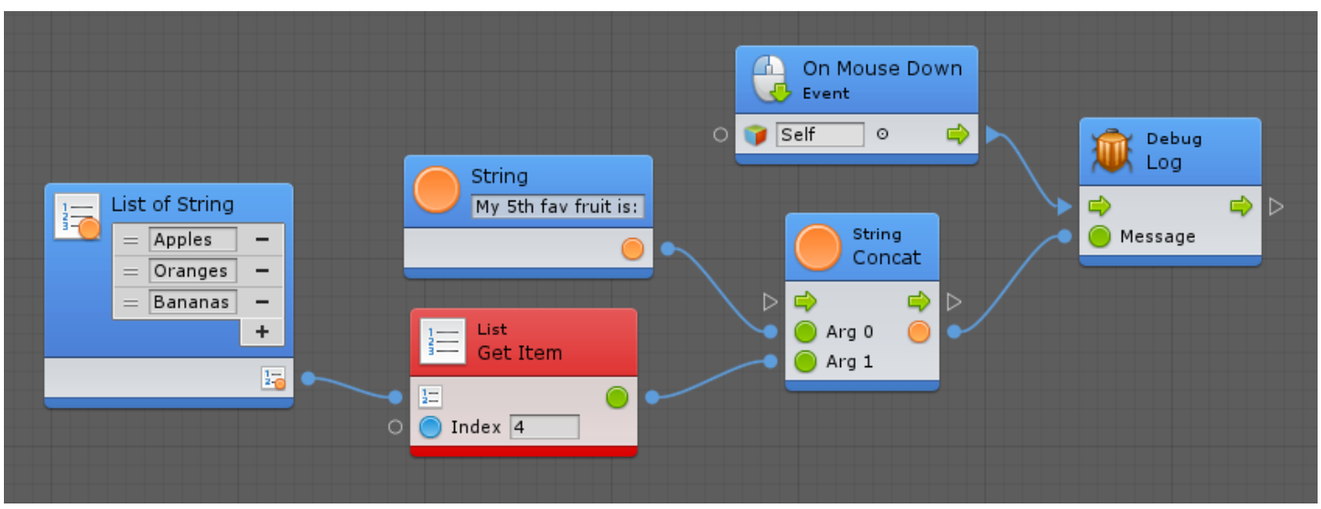
When in play mode, the currently active nodes are highlighted in **blue**.

If an error occurs, the node that caused it will be highlighted in **red**.

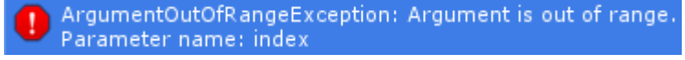
Let's take a look at a faulty graph. Here, we're trying to log our fifth favorite fruit to the console when we click the object. (*Surely this is an example that could apply to a game... right?*)



Can you guess what will go wrong? If we press play and click the object, here's what happens:



As you can see, all nodes are highlighted in blue as soon as we click because they were activated. However, there was an error in the console. Uh oh!



That's fine, but it doesn't tell us exactly where we went wrong. To make our life easier, Bolt highlighted the faulty node in red.

When you look at the graph, the error is now obvious: we're trying to get the 5th item in our list of strings, but we only have 3 items in the list. Hence why Unity complains that our argument is *out of range*.

> Why are we writing **4 for index** if we want to get the **5th item**? Because in scripting, **indices are always zero-based**. That means the first item is at index 0, the second at index 1, the third at index 2, etc.

If we add enough items to our list, the graph works as expected: