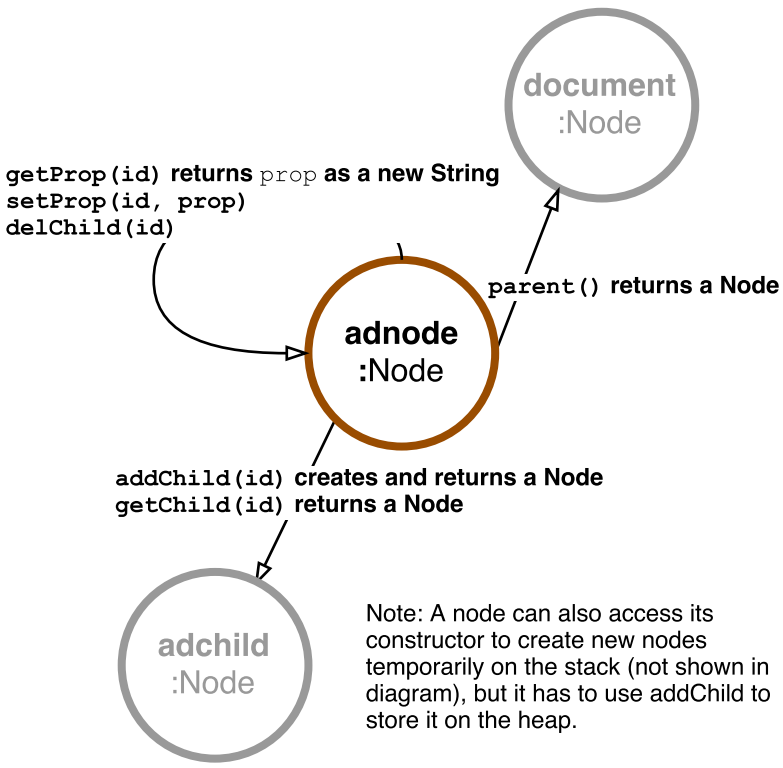
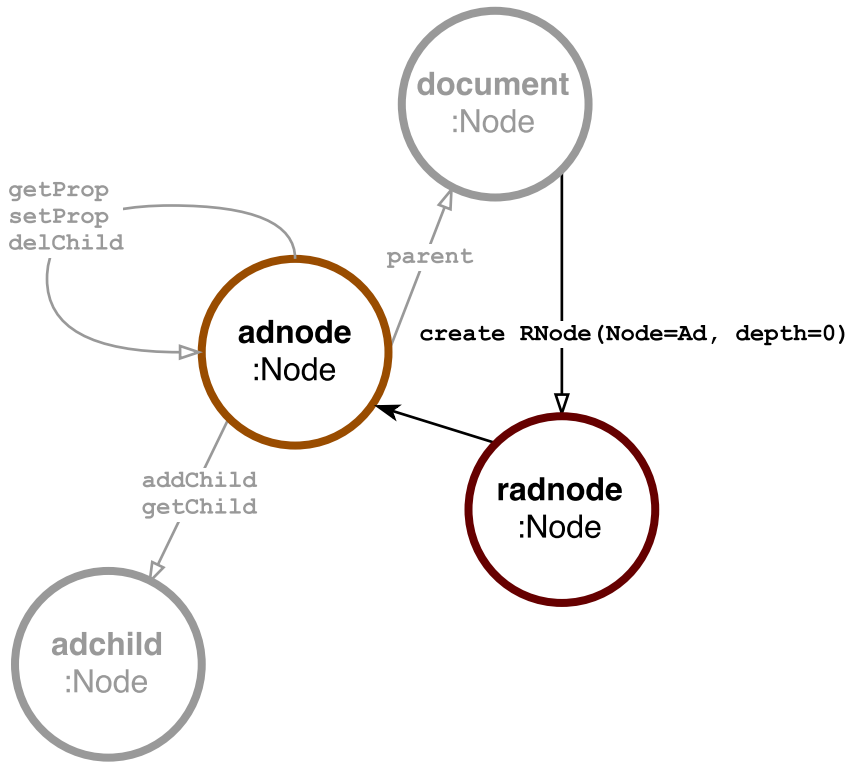
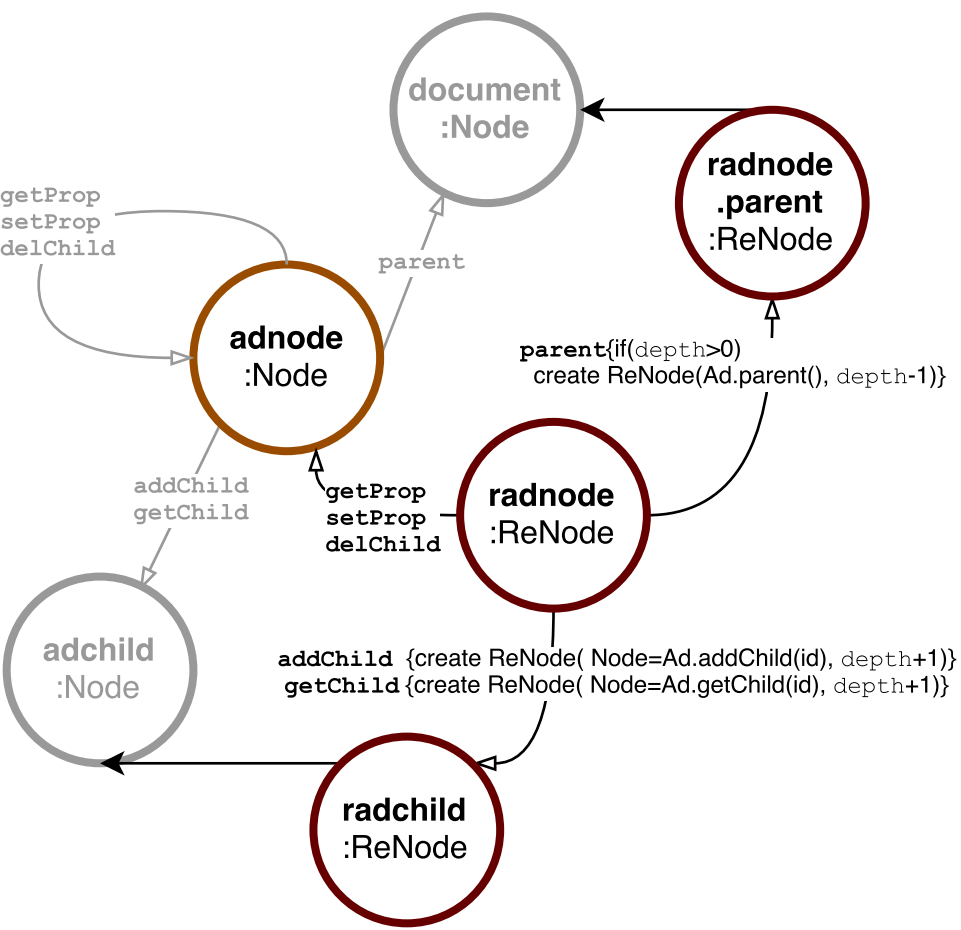
1) A simplified representation of a Javascript HTML DOM tree. A node can perform 6 functions and the result of each function call is pointed to by empty arrowheads. Notice below that giving away the capability of **adnode** to a third-party is unsafe, because using the parent() function call on the **adnode** returns **document,** the root node, from which all nodes and their capabilities in the entire DOM tree can be accessed.



2) A `Node` can now construct an attenuating `ReNode` over a child `Node` it has created, and also specify an integer variable `depth` to restrict how far up in a tree the newly created `ReNode` can travel. A `ReNode` with `depth=0` means that it cannot access its immediate parent. Also, `depth` can only be declared once in the `ReNode` constructor and cannot be subsequently changed or re-declared (`depth` is of a Javascript `let` type). The `ReNode` possesses the capability of the `Node` that it wraps over (filled arrowhead in diagram below) but this is stored in a private field. Therefore the capability of `Node` is not accessible externally and can only be used internally by `ReNode`'s functions.



3) A `ReNode` has all the functions of a `Node`, and it forwards all capability-insensitive messages (that return a non-capability type - `getProp`, `setProp`, `delChild`) to the `Node` that it wraps over , and returns `Node`'s results. For capability-sensitive functions that return a capability (`addChild`, `getChild`, `parent`), `ReNode` always checks some condition and if successful, always creates and return a new `ReNode` imbued with an adjusted `depth` so as to protect the access integrity of the tree. The function `parent` succeeds only if the `ReNode` has sufficient depth access to call its immediate parent (`depth>0`).



4) In the final diagram below, notice how it is safe now to give away the capability of the `RNode` **RAd** to a third-party, when **RAd** is constructed by **Document** with `depth=0`. The wrapper guarantees that the user of **RAd** cannot modify the properties of **Document** through the chained function call `parent().setProp(id, prop)` because `parent()` will first fail. Consequently, the wrapper also prevents **RAd**'s user from accessing any other node descended from **Document.**