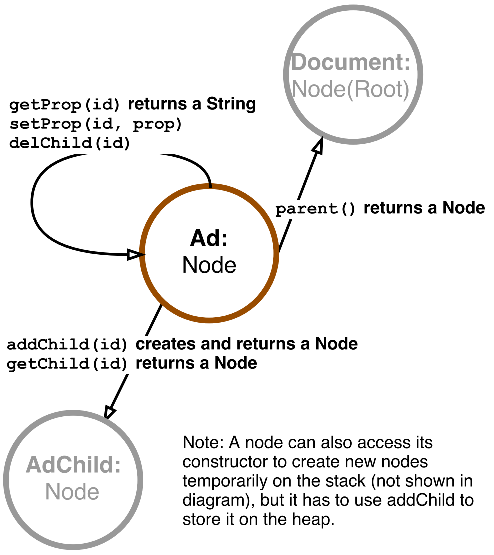
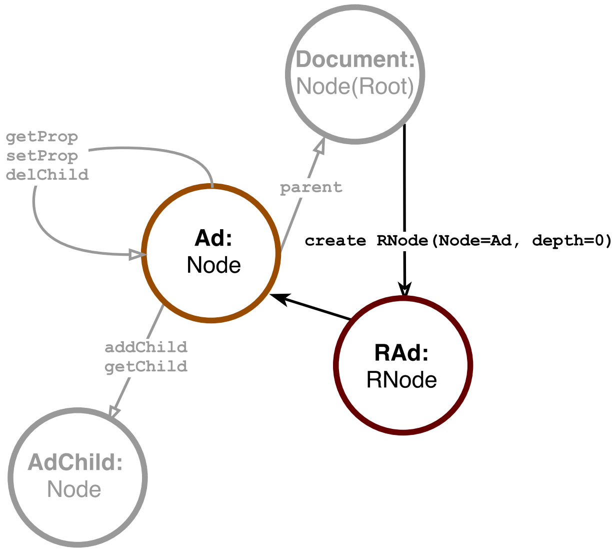
1) A simplified representation of a Javascript HTML DOM tree. A node can perform 6 functions and the result of each function call is pointed to by empty arrowheads. Notice below that giving away the capability of the **Ad** node to a third-party is unsafe, because using the `parent()` function call on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



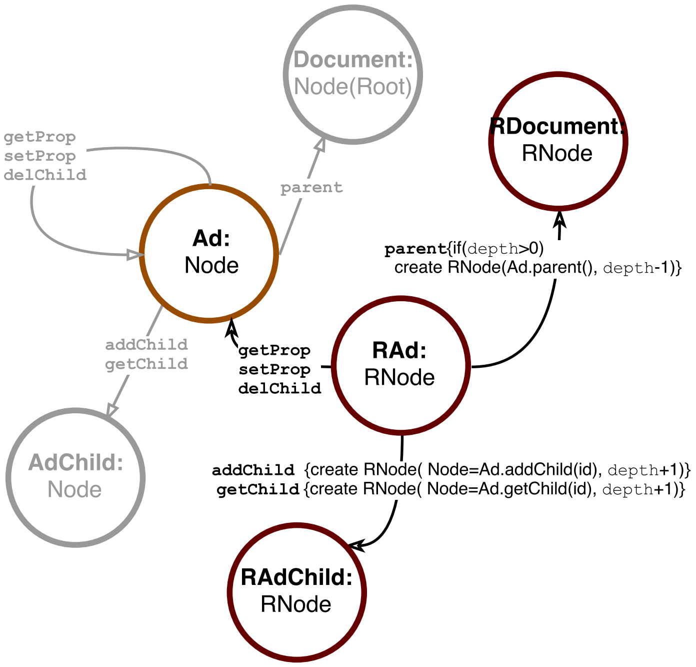
Document:
Node(Root)
getProp(id) returns a String
setProp(id, prop)
delChild(id)
parent() returns a Node
Ad:
Node
addChild(id) creates and returns a Node
getChild(id) returns a Node
AdChild:
Node

Note: A node can also access its constructor to create new nodes temporarily on the stack (not shown in diagram), but it has to use addChild to store it on the heap.

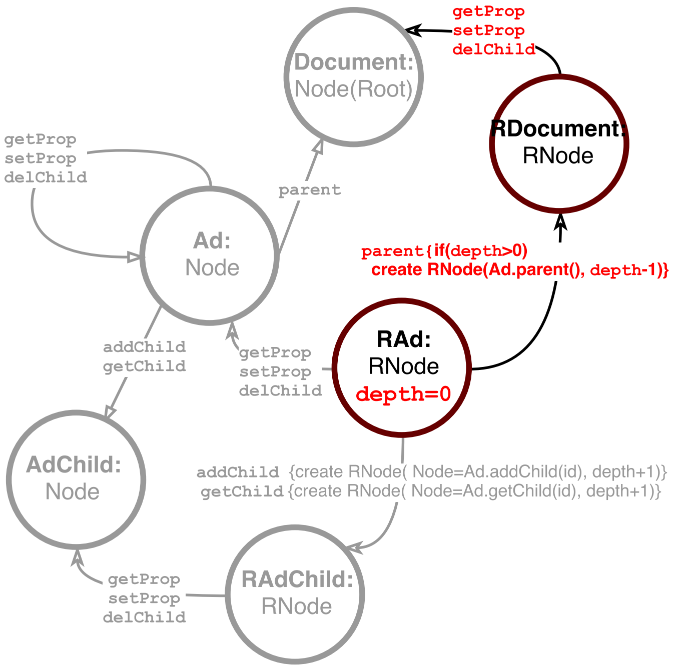
2) A `Node` can now construct a wrapper `RNode` over a child `Node` it has created, and also specify an integer variable `depth` to restrict how far up in a tree the newly created `RNode` can travel. A `RNode` with `depth=0` means that it cannot access its immediate parent. Also, `depth` can only be declared once in the `RNode` constructor and cannot be subsequently changed or re-declared (`depth` is of a Javascript `let` type). The `RNode` 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 `RNode`'s functions.



3) A `RNode` has all the functions of a `Node`, and it forwards all capability-insensitive functions (`getProp`, `setProp`, `delChild`) to the `Node` that it wraps over , and returns `Node`'s results. For functions that return a capability (`addChild`, `getChild`, `parent`), `RNode` always creates and return a new `RNode` with an adjusted `depth` to protect the access integrity of the tree. Moving up the tree results in a decremented `depth`, while moving down results in an incremented `depth`. In addition, the function `parent` checks if the `RNode` has sufficient depth access to call its immediate parent, and will throw an error if it does not.



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. The wrapper also prevents **RAd**'s user from accessing any other node descended from **Document.**



getProp
setProp
delChild
Document:
Node(Root)
RDocument:
RNode
getProp
setProp
delChild
parent
Ad:
Node
parent{if(depth>0)
create RNode(Ad.parent(), depth-1)}
addChild
getChild
getProp
setProp
delChild
RAd:
RNode
depth=0
AdChild:
Node
addChild {create RNode(Node=Ad.addChild(id), depth+1)}
getChild {create RNode(Node=Ad.getChild(id), depth+1)}
getProp
setProp
delChild
RAdChild:
RNode