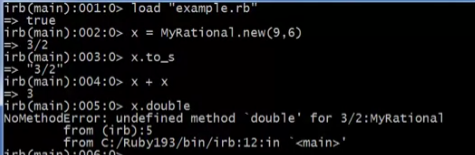
Changing classes

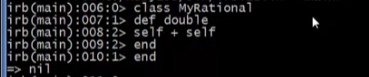
* Ruby programs (or the REPL) can add/change/replace methods while a program is running
* Breaks abstraction and makes programs very difficult to analyze, but it does have plausible uses
  + Simple example: Add a useful helper method to a class you did not define
    - Controversial in large programs, but may be useful
* For us: Helps re-enforce “the rules of OOP”
  + Every object has a class
  + A class determines its instances’ behavior

Example

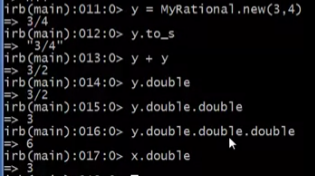
* Using our previous example, we do not have a double method in our code base:

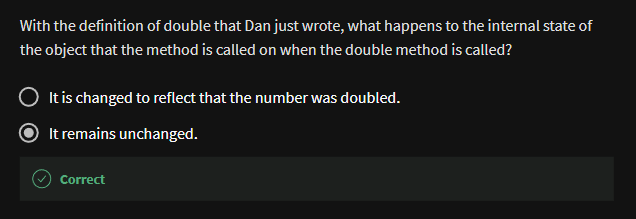


* But even if the codebase is not accessible, we can make a double method here in the REPL

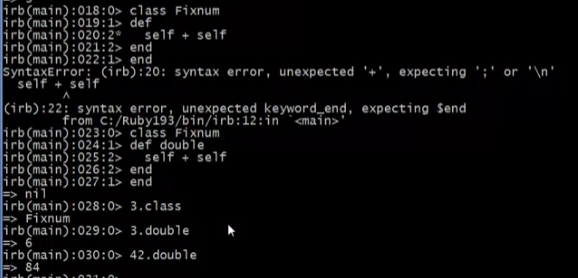


* We can now use our double method

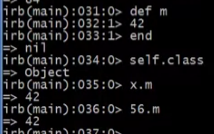




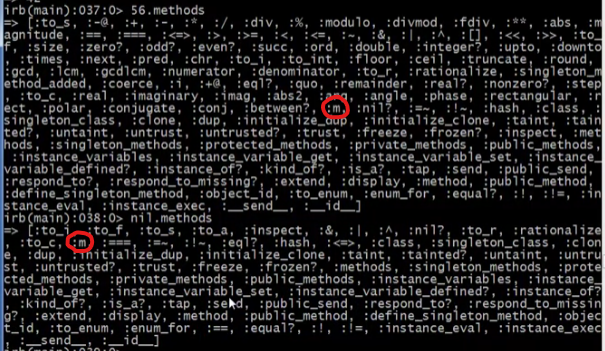
We can even define a method for the built-in class **Fixnum**



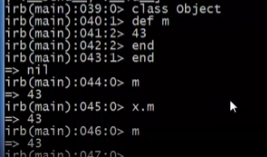
We can even define top-level method (in the **Object** class)



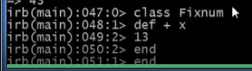
* m is added in the methods of all objects, even nil



* we can also replace the definition of our m



* But trying to replace the method body code for Fixnum (or any built-in class)



Will crash your REPL!!

The moral

* Dynamic features cause interesting semantic questions
* Example:
  + First create an instance of class C eg. x = C.new
  + Now replace method method m in C
  + Now call x.m

Old method or new method? In Ruby, new method

The point is Java/C#/C++ do not have to ask the question

* + May allow more optimized method-call implementation as a result