Basic Ruby:

Ques 1: Explanation about the MRI , JRUBY , RUBINIUS implementation.
Ans:

MRI: Matz's Ruby Interpreter or Ruby MRI (also called **CRuby**).
latest version is ruby 2.0.0.

JRUBY: JRuby is a ruby interpreter that gives the best of the JVM -- as battle-tested in the field for many years -- together with the awesomeness that is Ruby syntax.

RUBINIUS: Rubinius is a ruby interpreter written in C++ using many advanced techniques and technologies such as LLVM, a "precise, compacting, generational garbage collector",
a compatible C extension API to standard MRI Ruby, and many other awesome ideas that will give any language-nut a huge nerdon.

Differences Between MRI And JRuby:

Although ideally MRI and JRuby would behave 100% the same in all situations, there are some minor differences. Some differences are due to bugs, and those are not reported here.

a. Native C Extensions:

JRuby cannot run native C extensions. Popular libraries have all generally been ported to Java Native Extensions. Also, now that FFI has become a popular alternative to binding to C libraries, using it obviates the need to write a large chunk of native extensions.

b. Continuations and Fibers:

JRuby does not support continuations (Kernel.callcc).
Fibers (a form of delimited continuation) are supported on JRuby, but each fiber is backed by a native thread. This can lead to resource issues when attempting to use more fibers than the system allows threads in a given process.

c. Invoking external processes:

On Microsoft Windows, JRuby is a little smarter when launching external processes. If the executable file is not a binary executable (.exe), MRI requires you give the file suffix as well, but JRuby manages without it.
For example, say you have file foo.bat on your PATH and want to run it.
system('foo') # works on JRuby, fails on MRI
system('foo.bat') # works both in JRuby and MRI

d. Fork is not implemented:

JRuby doesn't implement fork() on any platform, including those where fork() is available in MRI. This is due to the fact that most JVMs cannot be safely forked.

e. Native Endian is Big Endian:

Since the JVM presents a *compatible* CPU to JRuby, the *native* endianness of JRuby is Big Endian. This does matter for operations that depend on that behavior, like String#unpack and Array#pack for formats like I, i, S, and s.

f. Time precision:

Since it is not possible to obtain usec precision under a JVM, Time.now.usec cannot return values with microsecond precision.

Example:

```
> Time.now.usec
=> 582000
```

Keep this in mind when counting on usec precision in your code.

g. Regular expressions:

JRuby only has one regular expression engine, which matches Onigurama's behavior. It is not changed in --1.8 mode, so code depending on regular expressions behaving precisely as on MRI 1.8.n may fail on JRuby in --1.8 mode. For instance:

```
ruby-1.8.7-p302 > "a".match(/^(.*)+$/)[1]
=> "a"
jruby-head > "a".match(/^(.*)+$/)[1]
=> ""
```

h. Thread priority:

In MRI, the Thread priority can be set to any value in Fixnum (if native threads are enabled) or -3..3 (if not). The default value is 0.

In JRuby, Threads are backed by Java threads, and the priority ranges from 1 to 10, with a default of 5. If you pass a value outside of this range to Thread#priority=, the priority will be set to 1 or 10.

Ques 2: SOLID principles of object oriented languages?

Ans:

In computer programming, SOLID (*Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*) is a mnemonic acronym introduced by Michael Feathers for the "first five principles" identified by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design. The principles when applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time. The principles of SOLID are guidelines that can be applied while working on software to remove code smells by causing the programmer to refactor the software's source code until it is both legible and extensible. It is typically used with test-driven development, and is part of an overall strategy of agile and adaptive programming.

1. Single responsibility principle:

a class should have only a single responsibility.

There should never be more than one reason for a class to change. Basically, this means that your classes should exist for one purpose only. For example, let's say you are creating a class to represent a

SalesOrder. You would not want that class to save to the database, as well as export an XML-based receipt. Why? Well if later on down the road, you want to change database type (or if you want to change your XML schema), you're allowing one responsibility's changes to possibly alter another. Responsibility is the heart of this principle, so to rephrase there should never be more than one responsibility per class.

2. Open/closed principle:

"software entities … should be open for extension, but closed for modification".

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. At first, this seems to be contradictory: how can you make an object behave differently without modifying it? The answer: by using abstractions, or by placing behavior(responsibility) in derivative classes. In other words, by creating base classes with override-able functions, we are able to create new classes that do the same thing differently without changing the base functionality. Further, if properties of the abstracted class need to be compared or organized together, another abstraction should handle this. This is the basis of the "keep all object variables private" argument.

3. Liskov substitution principle:

"objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. In other words, if you are calling a method defined at a base class upon an abstracted class, the function must be implemented properly on the subtype class. Or, "when using an object through its base class interface, [the] derived object must not expect such users to obey preconditions that are stronger than those required by the base class." The ever-popular illustration of this is the square-rectangle example. Turns out a square is not a rectangle, at least behavior-wise.

4. Interface segregation principle:

"many client-specific interfaces are better than one general-purpose interface."

Clients should not be forced to depend upon interfaces that they do not use. My favorite version of this is written as "when a client depends upon a class that contains interfaces that the client does not use, but that other clients do use, then that client will be affected by the changes that those other clients force upon the class." Kinda sounds like the inheritance specific single responsibility principle.

5. Dependency inversion principle:

one should "Depend upon Abstractions. Do not depend upon concretions."

Dependency injection is one method of following this principle.

Depend on abstractions, not on concretions or High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions. (I like the first explanation the best.) This is very closely related to the open closed principle we discussed earlier. By passing dependencies (such as connectors to devices, storage) to classes as abstractions, you remove the need to program dependency specific. Here's an example: an Employee class that needs to be able to be persisted to XML and a database. If we placed ToXML() and ToDB() functions in the class, we'd be violating the single responsibility principle. If we created a function that took a value that represented whether to print to XML or to DB, we'd be hard-coding a set of devices and thus be violating the open closed principle. The best way to do this would be to:

1. Create an abstract class (named DataWriter, perhaps) that can be inherited from for XML (XMLDataWriter) or DB (DbDataWriter) Saving, and then
2. Create a class (named EmployeeWriter) that would expose an Output(DataWriter saveMethod) that accepts a dependency as an argument. See how the Output method is dependent upon the

abstractions just as the output types are? The dependencies have been inverted. Now we can create new types of ways for Employee data to be written, perhaps via HTTP/HTTPS by creating abstractions, and without modifying any of our previous code! No rigidity--the desired outcome.

SRP	**The Single Responsibility Principle**	*A class should have one, and only one, reason to change.*
OCP	**The Open Closed Principle**	*You should be able to extend a classes behavior, without modifying it.*
LSP	**The Liskov Substitution Principle**	*Derived classes must be substitutable for their base classes.*
ISP	**The Interface Segregation Principle**	*Make fine grained interfaces that are client specific.*
DIP	**The Dependency Inversion Principle**	*Depend on abstractions, not on concretions.*

Ques 3: what is mean by symbol in RUBY ? what are the uses ?
Ans:
a symbol is something that you use to represent names and strings. What this boils down to is a way to efficiently have descriptive names while saving the space one would use to generate a string for each naming instance.

Using symbols not only saves time when doing comparisons, but also saves memory, because they are only stored once.
Symbols in Ruby are basically **"immutable strings"** .. that means that they can not be changed, and it implies that the same symbol when referenced many times throughout your source code, is always stored as the same entity, e.g. has the same object id.
Strings on the other hand are mutable, they can be changed anytime. This implies that Ruby needs to store each string you mention throughout your source code in it's separate entity, e.g. if you have a string "name" multiple times mentioned in your source code, Ruby needs to store these all in separate String objects, because they might change later on (that's the nature of a Ruby string).
If you use a string as a Hash key, Ruby needs to evaluate the string and look at it's contents (and compute a hash function on that) and compare the result against the (hashed) values of the keys which are already stored in the Hash.
If you use a symbol as a Hash key, it's implicit that it's immutable, so Ruby can basically just do a comparison of the (hash function of the) object-id against the (hashed) object-ids of keys which are already stored in the Hash. (much faster)
Downside: Each symbol consumes a slot in the Ruby interpreter's symbol-table, which is never released. Symbols are never garbage-collected. So a corner-case is when you have a large number of symbols (e.g. auto-generated ones). In that case you should evaluate how this affects the size of your Ruby interpreter.
Notes:
If you do string comparisons, Ruby can compare symbols just by their object ids, without having to evaluate them. That's much faster than comparing strings, which need to be evaluated.
If you access a hash, Ruby always applies a hash-function to compute a "hash-key" from whatever key you use. You can imagine something like an MD5-hash. And then Ruby compares those "hashed keys" against each other.

Ques 4: Draw the basic object hierarchy of ruby ?
Ans:

Objects

Everything is an object. You can't get very far into Ruby-land without hearing that phrase. It's true though, everything in Ruby is an object. The interesting thing is how those objects are linked together and classified.

Classes

All objects are instances of a class. There are three special classes that are the key to understanding the Ruby object model: Object, Module and Class. More on them later.

Relations

Like I said before, it's the links between the different objects that explain how the Ruby object model works.

Instance of/class

All objects have a class – they are *instances of* that class. That means that whenever you have a class, you can call .new on it to get an instance. And whenever you have an instance, you can call .class to get its class. This seems obvious, but it can get confusing when you remember that *everything is an object*, and so *everything is an instance of a class*.

Superclass/subclass

All classes have a link to a superclass. If you don't explicitly inherit from another class in a class definition, that class will inherit directly from the Object class.

How they fit together

Here's some code to play with:

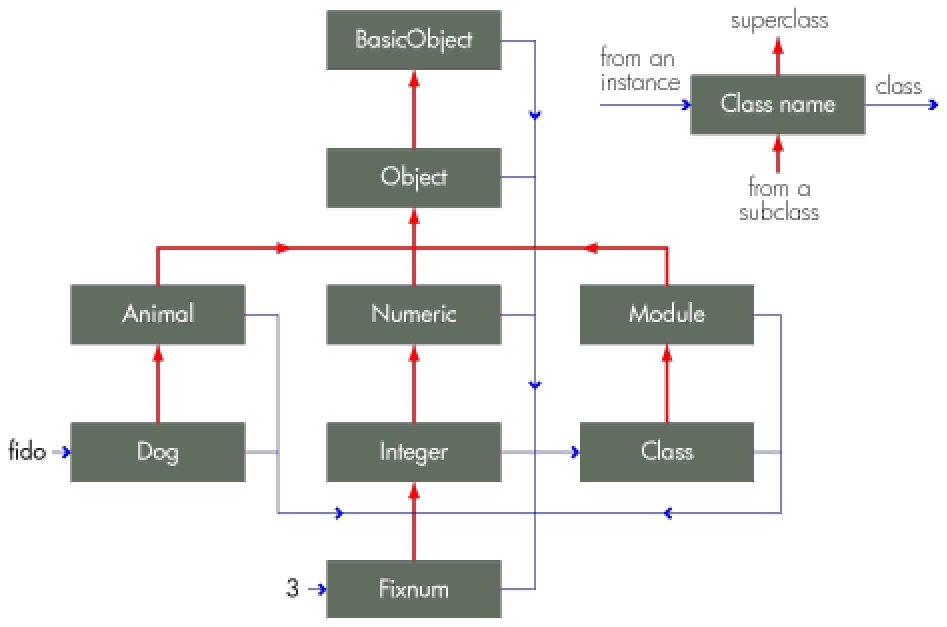
```
class Animal
end

class Dog < Animal
end

fido = Dog.new
```

Once you know what the pieces and the relations are, it's fairly straightforward to piece them together. All classes are an instance of Class. That includes the class Class, so there's a circular relation there (which isn't a problem).
Dog will basically have the following inheritance hierarchy (the superclass links):
Dog < Animal < Object < BasicObject

(Ruby 1.9 introduced BasicObject as Object's superclass.)
The Class class has the following inheritance hierarchy:

Class < Module < Object < BasicObject

Anything that's a class (so BasicObject, Object, Module, Class, Animal and Dog here) has Class as its class – they're all instances of Class. Any other objects have their own classes. So fido above will have a class of Dog, 3 has a class of Fixnum, "an awesome string" has a class of String, etc. All those classes inherit from Object ultimately (although classes like Fixnum have their own complex inheritance hierarchies). To finish things off, here's a little diagram of what I've just described, showing all the class and superclass links.



In next diagram, to find the class of any object, we traverse the red arrows. To find superclasses of class objects, we traverse the green arrows.

If a red arrow goes from a box A to a box B, then we say that object A is an instance of class B.

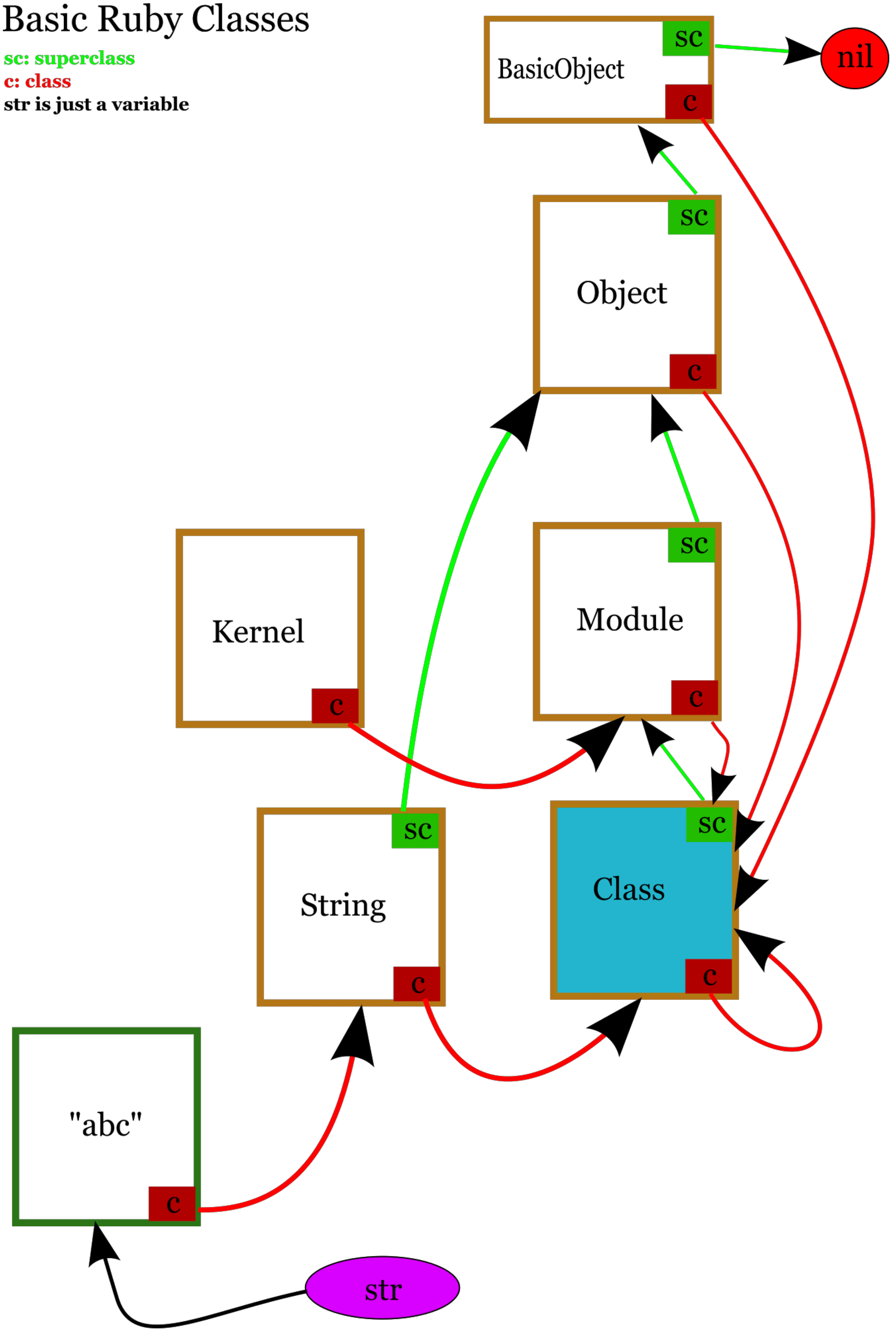
If a green arrow goes from a box A to a box B, then we say that class B is the superclass of class A.

Thus, the object Kernel is an instance of class Module and `str` is an instance of class String.

We use both "class" and "object" to refer to the boxes in this diagram and

Basic Ruby Classes

sc: superclass
c: class
str is just a variable



Ques 5: Write a program that will have three types of musicians which are Guitarist, Vocalist and drummer. All of them can have a method as playmusic method. There is a music band that will invoke all the types of musicians and it can have playmusic.

Ans:

for solution of this question, i have a different file(i.e. music.rb)