The Smalltalk Meta Object Protocol
(part I)

1. Which is the class all other classes in Smalltalk inherit from (directly or indirectly)?
   Object

2. Create an instance of that class.
   (a) What is the class of that instance?
      Object
   (b) What message can you send the instance to get the answer to (a)?
      The message `class` can be used for this. In this case by evaluating for example `Object new class`,
      which first creates a new instance of Object and then sends that instance the class message, resulting in
      the class Object.
   (c) How can you see the same in an inspector window without sending a message?
      The inspector window shows the class of the object it is inspecting in its title bar. When opening an
      inspector on the result of doing `Object new` the title bar reads 'an Object'.

3. What is the class of the class obtained in question 2.a?
   This can be determined by sending the message `class` to Object, which results in `Object class`.
   Note that when you open an inspector on Object, the title bar doesn't read 'an Object class' but just shows 'Core.Object'.

4. What is the class of the class obtained in question 3?
   This can be determined by evaluating `Object class class`, which first asks Object for its class,
   and then asks that class for its class. The result is the class Metaclass.

5. (a) What is the `superclass` of the class obtained in question 3?
   Class
   (b) What message can you send the class to get the answer to (a)?
      The message `superclass` can be used for this, for example by evaluating `Object class superclass`. 
6. Query the classes obtained in questions 2.a, 3 and 4 for their superclass. Then get the superclasses of those classes and so on until you can't go any further. Write down the complete superclass chain for each of the three classes from 2.a, 3 and 4:

Object

Object class => Class => ClassDescription => Behavior => Object

Metaclass => ClassDescription => Behavior => Object

Note: When you send the message superclass to Object you get nil meaning that Object has no superclass, not that nil is the superclass of Object.

7. (a) How many direct subclasses does the root class in Smalltalk have? 384 (the exact answer may depend on your specific version of VW, how many classes you've implemented yourself etc. ...)

(b) What message can you send the class to get the answer to (a)? The message subclasses can be used for this, it returns a collection of all the class's subclasses. This collection can then be queried for its size: Object subclasses size.

(c) What message can you send the class to get all subclasses (direct and indirect ones)? allSubclasses

(d) How many direct and indirect subclasses does the root class have? 4989, which can be determined by evaluating Object allSubclasses size. (again the exact answer may depend on your VW version etc.)

(e) How many classes are there in your Smalltalk image? 4990, which is the answer to the above question + 1, because Object itself is not included in the collection returned by allSubclasses. (again: exact answer may vary ...)

9. (a) How many instances does the class Array have? (Use the allInstances message)

(the answer to this question varies a lot depending on what you've been doing in the VW image, it's only important to grasp that it's probably more than 1)

(b) How many instances does the class of the class Array have? 1, as can be determined by evaluating Array class allInstances size

(c) How many instances does the class obtained in question 3 have? 1, as can be determined by evaluating Object class allInstances size
(d) What happens when you try to create a new instance of the class in question 3?
Evaluating `Object class new` results in an error message saying that 'a Metaclass should only have one instance'.

(e) How many instances does the class obtained in question 4 have?
2495, as can be determined by evaluating `Metaclass allInstances size` again.

(f) Create a new class. What happens to the answers for question 7.e and 9.e?
The result of `Object allSubclasses size` increases by two, while the result of `Metaclass allInstances size` increases by one.

(g) What is the relationship between the number from 9.e and the one from question 7.e? How does this relationship arise?
2 * 2495 = 4990 (the answers to 9.e and 7.e may differ in your image, but the relationship that one is twice the other should hold)

There are 2495 'regular' classes in the Smalltalk image, these are the classes you see in the class browser. The other 2495 classes are the meta classes of the 'regular' classes, such as 'Object class' which is the meta class of 'Object'. One such meta class is automatically created for each newly defined 'regular' class, thus increasing the number of meta classes by one, and the total number of classes by 2 (the newly created class and its meta class). The reason `Object allSubclasses size` does not just count the 'regular' classes but also the meta classes is because they are in fact subclasses of `Object` as can be seen in the answer to question 6: 'Object class' is a subclass of 'Class' which in turn is indirectly a subclass of 'Object'. (The meta class hierarchy also mimics the 'regular' class hierarchy, thus every meta class is (indirectly) a subclass of the 'Object class' meta class)

10. Make a drawing depicting the relationships between all the classes from question 6 as well as the classes `Collection` and `Bag`. Remember there are two different relationships between them: `is-instance-of` and `is-subclass-of` (use different colors).

This drawing is included on the last page of these solutions.

11. In questions 2 to 4 you had to get the class of a class, then the class of that class etc. How long can you go on doing this? What happens? Why is this? (Also: add the relationship you discover here to your diagram for question 10 if it's not on there already)

You could go on doing this *ad infinitum*: sending `class` to `Metaclass` results in its meta class, sending `class` to that again results in `Metaclass` and so on ... There is thus a small loop in the instance-of relationship between `Metaclass` and `Metaclass` class: each is an instance of the other!
12. (a) Send the message *selectors* to the class *OrderedCollection*. Explain the result:

   The result is a collection of the selectors of all the (instance) methods implemented by *OrderedCollection*.

(b) What do you get when you send the same message to the *class* of *OrderedCollection*? How is this related to what you view in the class browser in the user interface?

   The result is a collection of the selectors of all the (instance) methods implemented by *OrderedCollection* class. When opening the class browser on *OrderedCollection* the methods with the same selectors are shown in the tab 'Class Methods'. The distinction the class browser makes between 'instance methods' and 'class methods' isn't really there: the 'class methods' for a class are really just the (instance) methods of the meta class of that class.

13. You can send the message *new* to the root class even though it has no method for it.

(a) First verify that the class really doesn't have the method. Where do you need to look?

   When sending the message *new* to *Object*, the method lookup will begin in the class of *Object*, which is *Object* class. The methods that are implemented there can be viewed in the class browser by looking at the 'class methods' of *Object*, and these do not include a method for *new*.

(b) Which *new* method is executed when you send the message, where is it implemented? Why that one?

   Because there is no *new* method in *Object* class, the method lookup will continue in its superclass, which is *Class*. As can be seen by looking at the 'instance methods' (!!!) of *Class* in the class browser, there is no *new* method there either. Thus the method lookup will go with the superclass of *Class*, which is *ClassDescription* which also does not have a *new* method. The next superclass is *Behavior*, which does have a *new* method. Thus the method that is executed is the *new* method in *Behavior*.

(c) Repeat 9.d again, which *new* method is being executed here? Why that one? What does the method do?

   When sending a *new* message to *Object* class, the method lookup will begin in the class of *Object* class, which is *MetaClass*. This class overrides the *new* method from *Behavior* to generate the error message when the meta class already has an instance.

14. Send the messages *instVarAt: 4* and *instVarAt: 1* to *Collection*, what are the results? Why do you get these results? Can you send the *instVarAt:* message to all objects (try today's *Date* object) and why (not)?

   The *instVarAt: n* message can be sent to all objects and simply returns the value of the *n*th instance variable of the object. Which instance variable the *n*th instance variable is depends on the order in which the instance variables are defined in the class of the object and in the superclasses of that class: the list of instance variables an object has is a concatenation of the instance variables
defined in those classes, starting from the root class and then all the way down
the superclass chain to the class of the object itself. In the case of Collection,
whose class is Collection class, this means its first instance variable is the first
instance variable defined in Behavior (because Object doesn’t define instance
variables), which is superclass, the fourth is subclasses. Thus sending instVarAt:
1 to Collection gives its superclass, which is Object, and sending instVarAt: 4
gives its subclasses.

15. What happens when you send the message perform: #selectors to a class?
What other methods with perform: as the first keyword exist and what are they
used for?
Sending perform: #selectors to say Object, returns the same result as simply
sending selectors to Object. The perform: message can be sent to an object and
takes as argument a symbol and then sends the object a message with no
arguments and that symbol as the selector of the message. Variants of perform:
exist to send messages with arguments, such as the perform:with: message
which can for example be used as in 2 perform: #+ with: 3. You can find these
variants by using the Browse > Implementors of Selector command in the
Launcher.

16. Send the message methodDictionary to Date, what does the resulting object
represent? The object contains other objects, what are the classes of those
objects? And what do those objects contain?
The resulting object is an instance of MethodDictionary, it is a dictionary with as
values all the methods implemented in Date and as keys the selectors of those
methods. The selectors are instances of ByteSymbol, and the methods are
instances of CompiledMethod. The latter objects have three instance variables,
bytes, mclass and source. The source variable contains some number (this is
not important for this exercise), mclass contains the class in which the method
was defined (always Date in this case) and bytes contains the byte codes of the
method stored as an instance of ByteArray or SmallInteger for really short
methods. (The byte codes can be inspected in the inspector in a readable format
by looking at '-bytecodes', note that this is not actually an instance variable of the
methods though, it's just an extra convenience shown by the inspector.)

17. Implement a class with the following two methods. Create an instance of the
class and send it the methodOne message.

```
methodOne
    self methodTwo

methodTwo
    thisContext inspect.
    self halt.
```

What does thisContext contain? There's at least one tool in Smalltalk that is
implemented using this reflective mechanism, which one you think and why?
The thisContext special variable contains an instance of the class
MethodContext which represents the context in which the method is executed:
the most interesting instance variable it has is sender, which contains another
MethodContext instance with the context of the method that invoked the other
one. The thisContext variable thus allows you to reflect on the current method
call stack. This mechanism is used by the debugger tool in VisualWorks to allow you to trace through the call stack.

18. To the class you created in the previous question you should now add a method with selector `doesNotUnderstand:` with an empty body. Send the message `doSomethingCompletelyRandom` to an instance of the class. What would you normally expect to happen? What happens now? Why? Where is the "normal" behavior implemented?

The class does not implement a method for the selector `doSomethingCompletelyRandom`, nor does any of its (indirect) superclasses, thus when you send an instance of that class the message `doSomethingCompletelyRandom` it does not know how to respond to that message and you would expect an error dialog to pop up. However when you've implemented a `doesNotUnderstand:` method with an empty body, no error dialog pops up. The `doesNotUnderstand:` message is a message sent by the Smalltalk evaluator when the method lookup fails. The class `Object` provides a method for this message which pops up the error dialog you normally get.

19. Send a message `compile:` with an appropriate argument to the class you created in the previous questions. What would an appropriate argument be? What happens?

An appropriate argument for the `compile:` message is a string containing a method expression, because when a class is sent `compile:` it simply compiles that expression and adds the resulting method to its method dictionary.