New comparison operators

Description

\begin{verbatim}
=x <=> y returns -1, 0, 1 or nil
(x <=> y) == -1 means "x less than y" and is the same as x < y
(x <=> y) == 0 means "x equal y" is the same as x == y
(x <=> y) == 1 means "x greater than y" and is the same as x > y
(x <=> y) == nil means "x not comparable with y" and is the same as !(x <=> y)
We see there is no short syntax to test if two objects are not comparable.
Can we have something like (x >< y), provided by (Comparable) module, that would mean "x and y are not comparable"?
As ((x != y)) is a short syntax for (x != y)), can we have (x !< y)) for (x > y) and (x !<= y)) for (x >= y)) and so on?
As ((x <= y))) is the same as (x < y or x == y)), can we have (x <> y)) for (x < y or x > y))?
Here is a list of all possible comparison operators there could be (and their meanings):
< return true when <=< return -1 ("less than")
== return true when <=< return 0 ("equal")
<=< return true when <=< return 1 ("greater than")
= return true when <=< return 1 or 0 ("equal or greater than")
= return true when <=< return 1 or -1 ("less or greater than" that is to say "comparable and different")
< return true when <=< return nil ("not comparable")
<= return true when <=< return 0 or 1 or nil ("not less than", that is different from "greater than or equal" that involves "comparable")
= return true when <=< return 1 or 1 or nil ("not equal", that is different from "less or greater than" that involves "comparable")
> return true when <=< return 0 or 1 or nil ("not greater than", that is different from "less than or equal" that involves "comparable")
>= return true when <=< return nil or 1 or nil ("not equal nor less than", that is different from "greater than" that involves "comparable")
>= return true when <=< return nil or 0 or nil ("not equal nor greater than", that is different from "less than" that involves "comparable")
<> return true when <=< return 0 or nil ("not equal nor greater than", that is different from "equal" that involves "comparable")
<= return true when <=< return 0 or 1 ("not not comparable" or just "comparable", the same as <=< but returning a boolean)
\end{verbatim}

All these operators would be very useful, especially when working with partial orders (like subset-inclusion for example).

Perhaps ((x != y))) should also be a short syntax for (x != y)).
Imagine objects that are naturally ordered like in a tree structure. This is typically a partial order.

You would like to write \((\{a < b\})\) to test if \(a\) is a descendant of \(b\), \((\{a > b\})\) if \(a\) is an ancestor of \(b\), of course \((\{a == b\})\) if \(a\) and \(b\) are in the same position in the tree, but for two elements that are not in such a relationship, like siblings, you have to write \((\{!(a == b) \&\& !(a < b) \&\& !(a > b)\})\) or just \((\{!(a <= b) == nil\})\) or in the shorter way \((\{!(a <= b)\})\) that is still quite a long expression.

As \(<\), > and == operators are quick ways to test -1, 1 and 0 values of \(<\>, I think it would be fine to have such a way to test if \(a\) and \(b\) are not comparable, that is why I suggest to introduce the \(<>\) operator.

Then I thought that the Ruby principle of negative operators like !> and !<, that are not real methods but shortcut to express \((\{!(a == b)\})\) or \((\{!(a <= b)\})\), could be extended for all comparison operators.

With a total order, operators like \(<\) are not very useful because \(>=\) can do the job. But with a partial order, \(<\) and \(>=\) are quite different.

In my example of tree structure, a \(<\) \(b\) would express that \(a\) is not a child of \(b\). It can be an ancestor, but also a sibling.

Idem for \(<>\) operator : with a total order, \(!=\) do the job, but with partial order, \(<\) and \(!=\) have two different meanings.

I think those new operators would be in the Ruby way, and adding them would generalize what already exists in ruby. But I am aware that in most cases, we are working with total order relations for which half of those operators are redundant.

I hope I convinced you...

jEFF

---

#4 - 06/08/2012 01:11 AM - matz (Yukihiro Matsumoto)

One correction: you assume \((a != b)\) being equal to \(!(a == b)\), but since 1.9, \((a != b)\) calls \(!=\) method.

I understand your intention in certain degree. But sill in doubt to add operator.

Isn't it sufficient to add a few methods, not operators?

Matz.

#5 - 06/08/2012 09:19 PM - jEFF (Jean-François Berroyer)

I understand your hesitation in adding 8 more operators. It is not a little change. I am also aware that in Ruby, we can do without operators since methods can fit everywhere we use operators.

I really appreciate that in Ruby, operators are methods with just an alternative syntax, and that there is not another concept behind them, because there is no need. What I like with operators is that they make expressions close to mathematical notation, and often, mathematical notation is close to the concept it express.

When you write an equality with \((\{a == b\})\), you can see graphically that the relation is symetric, much more than in expressions like \((\{a.eql?(b)\})\). You can also see the antisymmetry in expressions like \((\{a < b\})\) or like \((\{!(a <= b)\})\).

The \((\{!\})\) prefix before an operator is for me just like a strike through the operator, like we do in mathematical notation.

Moreover, we are often hesitating on names when using methods rather than operators. \((\{equal?\})\), \((\{less_than?\})\), \((\{greater_than?\})\) etc. are often too long, especially when used in complex expressions. We can use abbreviations like \((\{eq\})\), \((\{gt\})\) or \((\{lt\})\). but doing this, the code is often less easy to read and to write (What does this abbreviation mean? What is the abbreviation for that?). That's also why I like operators. Their meanings are immediate because they are graphical and not linked with a language like english.

But I'm also very careful in using them, and I admit that very often, defining a method is a better choice than defining an operator. However, operators are for me the best solution for Comparable module.

I found it wonderful that the Comparable module provide all the comparison operators as soon as we have defined the \((\{<=>\})\) method (one to rule them all). But I don't like to use directly \((\{<=>\})\) in my code because its return values -1, 0, 1 and nil are just conventional (even if they are logical).

When I look at the documentation of Comparable module about \((\{<=>\})\) method, I have the impression that nil value is not considered like the other return values, but more like an exceptional one. I think it wouldn't be.

The concept of "comparable" in Ruby is for me directly connected with the mathematical concept of "order". In mathematics, there are partial orders and total orders, and a total order is just a particular partial order. So partial orders are more general than total orders. I think current Comparable module is perfect to work with total orders, but there are some missing things to work with partial orders, even if \((\{<=>\})\) returns four different values. Adding \((\{<\})\) and \((\{>=\})\) methods would help us to work with partial orders by getting the Comparable module more general.

I had a little hesitation with \((\{<\})\) and \((\{>=\})\), that would be new conventional notations. I hesitated in inventing the meaning of those two, but since in some other languages \((\{<\})\) is used to express that two things are different, I think it is ok if \((\{<\})\) means in Ruby "comparably different". About the \((\{<\})\) notation for "uncomparable", I don't have any strong argument, but I would just note that it doesn't need another character than those that are currently used in comparison operators.

03/18/2020
I also think that systematic use of \(((!!))\) prefix to have the negative form of a comparison operator is simple and logical and would also be a generalisation of what already exists.

Finally, I would like to note that all these new operators would not change anything in the current Ruby way: as long as you work with a total order, like with numbers natural order, or just with equivalence relations, there is no need to care about the new operators and you can programme just like before. But if you have to work with partial order, you will find those new operators very useful.

The only bad thing I think about is that introducing these new operators would involve that we will be able to define a lot of new operators in any class, and I think using them without Comparable module and giving them other arbitrary meanings could be a real bad thing. But is it Ruby's care to prevent users from doing bad things with Ruby?

The decision of adding all that stuff in Ruby is not a little one, because adding operators involves add-ons in Ruby's syntax. This idea has to be discussed, but I think it could be really interesting and that is why I wanted to propose it to the community.

jEFF
=end

#6 - 06/15/2012 07:57 PM - jEFF (Jean-François Berroyer)
=begin
A friend told me that D language provides 14 comparisons operators for Floating point comparisons: \(((URL:http://dlang.org/expression.html#RelExpression))\)

I don't know how really useful it is in D, but it is interesting because that is the same problem.

jEFF
=end

#7 - 10/27/2012 07:44 AM - ko1 (Koichi Sasada)
- Assignee set to matz (Yukihiro Matsumoto)

#8 - 10/27/2012 02:02 PM - matz (Yukihiro Matsumoto)
- Status changed from Open to Rejected

I am still in doubt for adding more comparison operators. We need clear benefit to make a big change like this.

Matz.