GAP

Release 4.5
May 2012

Migrating to GAP 4

The GAP Group

http://www.gap-system.org

Contents

1 Migrating to GAP 4

This chapter is intended to give users who have experience with GAP 3 some information about what has changed in GAP 4. In particular, it informs about changed command line options (see 1.1), the new global variable `fail` (see 1.2), some functions that have changed their behaviour (see 1.3) or their names (see 1.4), and some conventions used for variable names (see 1.5).

Then the new concepts of GAP 4 are sketched, first that of mutability or immutability (see 1.6), with the explanation of related changes in functions that copy objects (see 1.7), then the concepts of operations and method selection, which are compared with the use of operations records in GAP 3 (see 1.8, 1.10, and 1.11).

More local changes affect the concepts of notions of generation (see 1.9), of parents (see 1.12), of homomorphisms (see 1.13, 1.14, and 1.15), how elements in finitely presented groups are treated (see 1.16), how information about progress of computations can be obtained (see 1.18), and how one gets information in a `break` loop (see 1.19).

1.1 Changed Command Line Options

In GAP 4, the `-l` option is used to specify the **root directory** (see "GAP Root Directory" in the GAP 4 Reference Manual) of the GAP distribution, which is the directory containing the `lib` and `doc` subdirectories. Note that in GAP 3 this option was used to specify the path to the `lib` directory.

The `-h` option of GAP 3 has been removed, the path(s) for the documentation are deduced automatically in GAP 4.

The option `-g` is now used to print information only about full garbage collections. The new option `-g -g` generates information about partial garbage collections too.

1.2 Fail

There is a new global variable

1 ► `fail`

in GAP 4. It is intended as a return value of a function for the case that it could not perform its task. For example, `Inverse` returns `fail` if it is called with a singular matrix, and `Position` returns `fail` if the second argument is not contained in the list given as first argument.

GAP 3 handled such situations by either signalling an error, for example if it was asked for the inverse of a singular matrix, or by (mis)using `false` as return value, as in the example `Position`. Note that in the first example, in GAP 3 it was necessary to check the invertibility of a matrix before one could safely ask for its inverse, which meant that roughly the same work was done twice.

1.3 Changed Functionality

Some functions that were already available in GAP 3 behave differently in GAP 4. This section lists them.

1 ► `Orbit(` *G*`,`*pnt*`)`

The GAP 3 manual promised that *pnt* would be the first entry of the resulting orbit. This was wrong already there in a few cases, therefore GAP 4 does not promise anything about the ordering of points in an orbit.

2 ► `Order(` *g* `)`

only takes the element *g* and computes its multiplicative order. (Note that it does not make sense anymore to specify a group as first argument w.r.t. which the order of the second argument shall be computed, see 1.16.)

3 ► `Position(` *list*`,` *obj* `)`

If *obj* is not contained in the list *list* then `fail` is returned in GAP 4 (see 1.2), whereas `false` was returned in GAP 3.

4 ► `PermGroupOps.ElementProperty(` *G*`,` *prop*`[,` *K*`] )`

In GAP 3, this function took either two or three arguments, the optional argument *K* being a subgroup of *G* that stabilizes *prop* in the sense that for any element *g* in *G*, either all elements or no element in the coset *g* `*` *K* have the property *prop*.

The GAP 4 function `ElementProperty`, however, takes between two and four arguments, and the subgroup *K* known from GAP 3 has to be entered as the **fourth** argument **not** the third. (The third argument in the GAP 4 function denotes a subgroup *U* stabilizing *prop* in the sense that either all elements or no element in right cosets *U* `*` *g* have the property *prop*.)

(This discrepancy was discovered only in March 2002, short before the release of GAP 4.3.)

5 ► `Print(` *obj*`, ... )`

Objects may appear on the screen in a different way, depending on whether they are printed by the read eval print loop or by an explicit call of `Print`. The reason is that the read eval print loop calls the operation `ViewObj` and not `PrintObj`, whereas `Print` calls `PrintObj` for each of its arguments. This permits the installation of methods for printing objects in a short form in the read eval print loop while retaining `Print` to display the object completely. See also Section “View and Print” in the GAP 4 Reference Manual.

(`PrintObj` is installed as standard method `ViewObj`, so it is not really necessary to have a `ViewObj` method for an object.)

6 ► `PrintTo(` *filename*`,` *obj*`, ... )`

In GAP 3, `PrintTo` could be (mis)used to “redirect” the text **printed** by a function (that is, **not** only the output of a function) to a file by entering the function call as second argument. This was used mainly in order to avoid many calls of `AppendTo`. In GAP 4, this feature has disappeared. One can use streams (see Chapter “Streams” in the GAP 4 Reference Manual) instead in order to write files efficiently.

1.4 Changed Variable Names

Some functions have changed their name without changing the functionality. A – probably incomplete – list follows

GAP 3	GAP 4	
AgGroup	PcGroup	# (also composita)
ApplyFunc	CallFuncList	
Backtrace	Where	
CharTable	CharacterTable	# (also composita)
Denominator	DenominatorRat	
DepthVector	PositionNonZero	
Elements	AsSSortedList	
IsBijection	IsBijective	
IsFunc	IsFunction	
IsMat	IsMatrix	
IsRec	IsRecord	
IsSet	IsSSortedList	
LengthWord	Length	
NOfCyc	Conductor	
Numerator	NumeratorRat	
NormedVector	NormedRowVector	
Operation	Action	# (also composita)
Order(G,g)	Order(g)	
OrderMat	Order	
OrderPerm	Order	
RandomInvertableMat	RandomInvertibleMat	
RecFields	RecNames	
X	Indeterminate	

1.5 Naming Conventions

The way functions are named has been unified in GAP 4. This might help to memorize or even guess names of library functions.

If a variable name consists of several words then the first letter of each word is capitalized.

If the first part of the name of a function is a verb then the function may modify its argument(s) but does not return anything, for example `Append` appends the list given as second argument to the list given as first argument. Otherwise the function returns an object without changing the arguments, for example `Concatenation` returns the concatenation of the lists given as arguments.

If the name of a function contains the word `By` then the return value is thought of as built in a certain way from the parts given as arguments. For example, `GroupByGenerators` returns a group built from its group generators, and creating a group as a factor group of a given group by a normal subgroup can be done by taking the image of `NaturalHomomorphismByNormalSubgroup` (see also 1.14). Other examples of "By" functions are `GroupHomomorphismByImages` and `UnivariateLaurentPolynomialByCoefficients`.

If the name of a function contains the word `Of` then the return value is thought of as information deduced from the arguments. Usually such functions are attributes (see "Attributes" in the GAP 4 Reference Manual). Examples are `GeneratorsOfGroup`, which returns a list of generators for the group entered as argument, or `DiagonalOfMat`.

For the setter and tester functions of an attribute *attr* (see 1.8 in this document and "Attributes" in the GAP 4 Reference Manual) the names `Set`*attr* resp. `Has`*attr* are available.

If the name of a function *fun1* ends with `NC` then there is another function *fun2* with the same name except that the `NC` is missing. `NC` stands for "no check". When *fun2* is called then it checks whether its arguments are valid, and if so then it calls *fun1*. The functions `SubgroupNC` and `Subgroup` are a typical example.

The idea is that the possibly time consuming check of the arguments can be omitted if one is sure that they are unnecessary. For example, if an algorithm produces generators of the derived subgroup of a group then it is guaranteed that they lie in the original group; `Subgroup` would check this, and `SubgroupNC` omits the check.

Needless to say, all these rules are not followed slavishly, for example there is one operation `Zero` instead of two operations `ZeroOfElement` and `ZeroOfAdditiveGroup`.

1.6 Immutable Objects

GAP 4 supports "immutable" objects. Such objects cannot be changed, attempting to do so issues an error. Typically attribute values are immutable, and also the results of those binary arithmetic operations where both arguments are immutable. For example, `[1..100]+[1..100]` is a mutable list and `2 * Immutable([1..100])` is an immutable list, both are equal to the (mutable) list `[2,4..200]`.

There is no way to **make** an immutable object mutable, one can only get a mutable copy by `ShallowCopy`. The other way round, `MakeImmutable` makes a (mutable or immutable) object and all its subobjects immutable; one must be very careful to use `MakeImmutable` only for those objects that are really newly created, for such objects the advantage over `Immutable` is that no copy is made.

More about immutability can be found in Sections "Immutability" in the GAP 4 Tutorial and "Mutability and Copyability" in the GAP 4 Reference Manual.

1.7 Copy

The function `Copy` of GAP 3 is not supported in GAP 4. This function was used to create a copy *cop* of its argument *obj* with the properties that *cop* and *obj* had no subobjects in common and that if two subobjects of *obj* were identical then also the corresponding subobjects of *cop* were identical.

The possibility of having immutable objects (see 1.6) can and should be used to avoid unnecessary copying. Namely, given an immutable object one needs to copy it only if one wants to get a modified object, and in such a situation usually it is sufficient to use `ShallowCopy`, or at least one knows how deep one must copy in order to do the changes one has in mind.

For example, suppose you have a matrix group, and you want to construct a list of matrices by modifying the group generators. This list of generators is immutable, so you call `ShallowCopy` to get a mutable list that contains the same matrices. If you only want to exchange some of them, or to append some other matrices, this shallow copy is already what you need. So suppose that you are interested in a list of matrices where some rows are also changed. For that, you call `ShallowCopy` for the matrices in question, and you get matrices whose rows can be changed. If you want to change single entries in some rows, `ShallowCopy` must be called to get mutable copies of these rows. Note that in all these situations there is no danger to change, i.e., to destroy the original generators of the matrix group.

If one needs the facility of the `Copy` function of GAP 3 to get a copy with the same structure then one can use the new GAP 4 function `StructuralCopy`. It returns a structural copy that has no **mutable** subobject in common with its argument. So if `StructuralCopy` is called with an immutable object then this object itself is returned, and if `StructuralCopy` is called with a mutable list of immutable objects then a shallow copy of this list is returned.

Note that `ShallowCopy` now is an operation. So if you create your own type of (copyable) objects then you must define what a shallow copy of these objects is, and install an appropriate method.

1.8 Attributes vs. Record Components

In GAP 3, many complex objects were represented via records, for example all domains. Information about these objects was stored in components of these records. For the user, this was usually not relevant, since there were functions for computing information about the objects in question. For example, if one was interested in the size of a group then one could call `Size`.

But since it was guaranteed that the size of a domain *D* was stored as value of the component `size`, it was allowed to access *D*`.size` if this component was bound, and a check for this was possible via `IsBound(` *D*`.size )`.

In GAP 4, only the access via functions is admissible. One reason is the following basic rule.

From the information that a given GAP 4 object is for example a domain, one cannot conclude that this object has a certain representation.

For attributes like `Size`, GAP 4 provides two related functions, the **setter** and the **tester** of the attribute, which can be used to set an attribute value and to check whether the value of an attribute is already stored for an object (see also "Attributes" in the GAP 4 Reference Manual). For example, if *D* is a domain in GAP 4 then `HasSize(` *D* `)` is `true` if the size of *D* is already stored, and `false` otherwise. In the latter case, if you know that the size of *D* is *size* then you may store it by `SetSize(` *D*, *size* `)`.

Besides the flexibility in the internal representation of objects, storing information only via function calls has also the advantage that GAP 4 is able to draw conclusions automatically. For example, as soon as it is stored that a group is nilpotent, it is also stored that it is solvable, see Chapters "Types of Objects" and "Method Selection" in the GAP 4 Reference Manual for the details.

As a consequence, you cannot put your favourite information into a domain *D* by assigning it to a new component like *D*`.myPrivateInfo`. Instead you can introduce a new attribute and then use its setter, see Section "Attributes" in the GAP 4 Reference Manual.

1.9 Different Notions of Generation

As in GAP 3, a **domain** in GAP 4 is a structured set.

The same set can have different structures, for example a field can be regarded as a ring or as an algebra or vector space over a subfield.

In GAP 3, however, an object representing a ring did not represent a field, and an object representing a field did not represent a ring. One reason for this was that the record component `generators` was used to denote the appropriate generators of the domain. For a ring *R*, the component *R*`.generators` was a list of ring generators, and for a field *F*, *F*`.generators` was a list of field generators.

GAP 4 cleans this up, see "Notions of Generation" in the GAP 4 Tutorial. It supports many different notions of generation, for example one can ask for magma generators of a group or for generators of a field as an additive group. A subtle but important distinction is that between generators of an algebra and of an algebra-with-one.

So the attributes `GeneratorsOfGroup`, `GeneratorsOfMagma`, `GeneratorsOfRing`, `GeneratorsOfField`, `GeneratorsOfVectorSpace`, and so on, replace the access to the `generators` component.

1.10 Operations Records

Already in GAP 3 there were several functions that were applicable to many different kinds of objects, for example `Size` could be applied to any domain, and the binary infix multiplication `*` could be used to multiply two matrices, an integer with a row vector, or a permutation with a permutation group. This was implemented as follows. Functions like `Size` and `*` tried to find out what situation was described by its arguments, and then it called a more specific function to compute the desired information. These more specific functions, let us call them **methods** as they are also called in GAP 4, were stored in so-called **operations records** of the arguments.

For example, every domain in GAP 3 was represented as a record, and the operations record was stored in the record component `operations`. If `Size` was called for the domain then the method to compute the size of the domain was found as value of the `Size` component of the operations record.

This was fine for functions taking only one argument, and in principle it is possible that for those functions an object stored an optimal method in its operations record. But in the case of more arguments this is not possible. In a multiplication of two objects in GAP 3, one had to choose between the methods stored in the operations records of the arguments, and if for example the method stored for the left operand was called, this method had to handle all possible right operands.

So operations records turned out to be not flexible enough. In GAP 4, operations records are not supported A detailed description of the new mechanism to select methods can be found in Chapter "Method Selection" in the GAP 4 Reference Manual.

An important point is that the new mechanism allows GAP to take the relation between arguments into account. So it is possible (and recommended) to install different methods for different relations between the arguments. Note that such methods need not do the extensive argument checking that was necessary in GAP 3, because most of the checks are done already by the method selection mechanism.

1.11 Operations vs. Dispatcher Functions

GAP 3 functions like `Size`, `CommutatorSubgroup`, or `SylowSubgroup` did mainly call an appropriate method (see 1.10) after they had checked their arguments. Such functions were called **dispatchers** in GAP 3. In GAP 4, many dispatchers have been replaced by **operations**, due to the fact that methods are no longer stored in operations records (see "Method Selection" in in the GAP 4 Reference Manual for the details).

Most dispatchers taking only one argument were treated in a special way in GAP 3, they had the additional task of storing computed values and using these values in subsequent calls. For example, the dispatcher `Size` first checked whether the size of the argument was already stored, and if so then this value was returned; otherwise a method was called, the value returned by this method was stored in the argument, and then returned by `Size`.

In GAP 4, computed values of operations that take one argument (these operations are called **attributes**) are also stored, only the mechanism to achieve this has changed, see the sections "Attributes" and "Properties" in the GAP 4 Reference Manual.

So the behaviour of `Size` is the same in GAP 3 and GAP 4. But note that in GAP 4, it is not possible to access *D*`.size`, see 1.8. As described in 1.10, GAP 4 does not admit "bypassing the dispatcher" by calling for example *D*`.operations.Size`. This was done in GAP 3 often for efficiency reasons, but the method selection mechanism of GAP 4 is fast enough to make this unnecessary.

If you had written your own dispatchers and put your own methods into existing operations records then this code will not work in GAP 4. See "Creating New Objects" and "Method Selection" in the GAP 4 Reference Manual for a description of how to define operations and to install methods.

Finally, some functions in GAP 3 were hidden in operations records, e.g., `PermGroupOps.MovedPoints`. These functions became proper operations in GAP 4.

1.12 Parents and Subgroups

In GAP 3 there was a strict distinction between parent groups and subgroups. The use of the name "parent" (instead of "supergroup") was chosen to indicate that the parent of an object was more than just useful information. In fact the main reason for the introduction of parents was to provide a common roof for example for all groups of polycyclic words that belonged to the same PC-presentation, or for all subgroups of a finitely presented group (see 1.16). A subgroup was never a parent group, and it was possible to create subgroups only of parent groups.

In GAP 4 this common roof is provided already by the concept of **families**, see Section "Families" in the GAP 4 Reference Manual. Thus it is no longer compulsory to use parent groups at all. On the other hand, parents **may** be used in GAP 4 to provide information about an object, for example the normalizer of a group in its parent group may be stored as an attribute value. Note that there is no restriction on the supergroup that is set to be the parent, it is possible to create a subgroup of any group, this group then being the parent of the new subgroup. This permits for example chains of subgroups with respective parents, of arbitrary length.

As a consequence, the `Parent` command cannot be used in GAP 4 to test whether the two arguments of `CommutatorSubgroup` fit together, this is now a question that concerns the relation between the families of the groups. So the 2-argument version of `Parent` and the now meaningless function `IsParent` have been abolished.

1.13 Homomorphisms vs. General Mappings

In GAP 3 there had been a confusion between group homomorphisms and general mappings, as `GroupHomomorphismByImages` created only a general mapping that did **not** store whether it was a mapping. This caused expensive, unwanted, and unnecessary tests whether the mapping was in fact a group homomorphism. Moreover, the "official" workaround to set some components of the mapping record was quite unwieldy.

In GAP 4, `GroupHomomorphismByImages` checks whether the desired mapping is indeed a group homomorphism; if so then this property is stored in the returned mapping, otherwise `fail` is returned. If you want to avoid the checks then you can use `GroupHomomorphismByImagesNC`. If you want to check whether a general mapping that respects the group operations is really a group homomorphism, you can construct it via `GroupGeneralMappingByImages` and then call `IsGroupHomomorphism` for it. (Note that `IsGroupHomomorphism` returns `true` if and only if both `IsGroupGeneralMapping` and `IsMapping` do, so one does in fact check `IsMapping` in this case.)

There is **no** function `IsHomomorphism` in GAP 4, since there are several different operations with respect to which a mapping can be a homomorphism.

1.14 Homomorphisms vs. Factor Structures

If F is a factor structure of G, with kernel N, complete information about the connection between F and G is provided by the **natural homomorphism**.

In GAP 3, the "official way" to construct this natural homomorphism was to create first the factor structure F, and then to call `NaturalHomomorphism` with the arguments G and F. For that, the data necessary to compute the homomorphism was stored in F when F was constructed.

In GAP 4, factor structures are not treated in a special way, in particular they do not store information about a homomorphism. Instead, the more natural way is taken to construct the natural homomorphism from G and N by `NaturalHomomorphismByNormalSubgroup` if N is a normal subgroup of the group G, or by `NaturalHomomorphismByIdeal` if N is an ideal in the ring G. The factor F can then be accessed as the image of this homomorphism, and of course G is the preimage and N is the kernel.

Note that GAP 4 does not guarantee anything about the representation of the factor F, it may be a permutation group or a polycyclically presented group or another kind of group. Also note that a natural homomorphism need not be surjective.

A consequence of this change is that GAP 4 does **not** allow you to construct a natural homomorphism from the groups G and F.

The other common type of homomorphism in GAP 3, "operation homomorphisms", have been replaced (just a name change) by **action homomorphisms**, which are handled in a similar fashion. That is, an action homomorphism is constructed from an acting group, an action domain, and a function describing the operation. The permutation group arising by the induced action is then the image of this operation homomorphism.

The GAP 3 function `Operation` is still supported, under the name `Action`, but from the original group and the result of `Action` it is not possible to construct the action homomorphism.

1.15 Isomorphisms vs. Isomorphic Structures

In GAP 3, a different representation of a group could be obtained by calling `AgGroup` to get an isomorphic polycyclically presented group, `PermGroup` to get an isomorphic permutation group, and so on. The returned objects stored an isomorphism in the record component `bijection`.

For the same reason as in 1.14, GAP 4 puts emphasis on the isomorphism, and the isomorphic object in the desired representation can be accessed as its image. So you can call `IsomorphismPcGroup` or `IsomorphismPermGroup` in order to get an isomorphism to a polycyclically presented group or a permutation group, respectively, and then call `Image` to get the isomorphic group.

Note that the image of an action homomorphism with trivial kernel is also an isomorphic permutation group, but an action homomorphism need not be surjective, since it may be easier to define it into the full symmetric group.

Further note that in GAP 3, a usual application of isomorphisms to polycyclically presented groups was to utilize the usually more effective algorithms for solvable groups. However, the new concept of polycyclic generating systems in GAP 4 makes it possible to apply these algorithms to arbitrary solvable groups, independent of the representation. For example, GAP 4 can handle polycyclic generating systems of solvable permutation groups. So in many cases, a change of the representation for efficiency reasons may be not necessary any longer.

In general `IsomorphismFpGroup` will define a presentation on generators chosen by the algorithm. The corresponding elements of the original group can be obtained by the command

```
gens:=List(GeneratorsOfGroup(Image(isofp)),i->PreImagesRepresentative(isofp,i));
```

If a presentation in the given generators is needed, the command `IsomorphismFpGroupByGenerators(`*G*`,` *gens*`)` will produce one.

1.16 Elements of Finitely Presented Groups

Strictly speaking, GAP 3 did not support elements of finitely presented groups. Instead, the "words in abstract generators" of the underlying free groups were (mis)used. This caused problems whenever calculations with elements were involved, the most obvious ones being wrong results of element comparisons. Also functions that should in principle work for any group were not applicable to finitely presented groups. In effect, a finitely presented group had to be treated in a special way in GAP 3.

GAP 4 distinguishes free groups and their elements from finitely presented groups and their elements. Comparing two elements of a finitely presented group will yield either the correct result or no result at all.

Note that in GAP 4, the arithmetic and comparison operations for group elements do not depend on a context provided by a group that contains the elements. In particular, in GAP 4 it is not meaningful to call `Order(` *G*`,` *g* `)` for a group *G* and an element *g*.

1.17 Polynomials

In GAP 3, polynomials were defined over a field. So a polynomial over `GF(3)` was different from a polynomial over `GF(9)`, even if the coefficients were exactly the same.

GAP 4 defines polynomials only over a characteristic. This makes it possible for example to multiply a polynomial over `GF(3)` with a polynomial over `GF(9)` without the need to convert the former to the larger field.

However it has an effect on the result of `DefaultRing` for polynomials: In GAP 3 the default ring for a polynomial was the polynomial ring of the field over which the polynomial was defined. In GAP 4 no field is associated, so (to avoid having to define the algebraic closure as the only other sensible alternative) the default ring of a polynomial is the `DefaultRing` of its coefficients.

This has an effect on `Factors`: If no ring is given, a polynomial is factorized over its `DefaultRing` and so `Factors(`*poly*`)` might return different results.

To be safe from this problem, if you are not working over prime fields, rather call `Factors(`*pring*`,`*poly*`)` with the appropriate polynomial ring and change your code accordingly.

1.18 The Info Mechanism

Sometimes it is useful to get information about the progress of a calculation. Many GAP functions contain statements to display such information under certain conditions.

In GAP 3, these statements were calls to functions such as `InfoGroup1` or `InfoGroup2`, and if the user assigned `Print` to these variables then this had the effect to switch on the printing of information. `InfoGroup2` was used for more detailed information than `InfoGroup1`. One could switch off the printing again by assigning `Ignore` to the variables, and `Ignore` was also the default value.

GAP 4 uses one function `Info` for the same purpose, which is a function that takes as first argument an **info class** such as `InfoGroup`, as second argument an **info level**, and the print statements as remaining arguments. The level of an info class *class* is set to *level* by calling `SetInfoLevel(` *class*`,` *level* `)`. An `Info` statement is printed only if its second argument is smaller than or equal to the current info level. For example,

```
gap> test:= function( obj )
> Info( InfoGroup, 2, "This is useful, isn't it?" );
> return obj;
> end;;
gap> test( 1 );
1
gap> SetInfoLevel( InfoGroup, 2 );
gap> test( 1 );
#I  This is useful, isn't it?
1
```

As in GAP 3, if an info statement is ignored then its arguments are not evaluated.

1.19 Debugging

If GAP 4 runs into an error or is interrupted, it enters a break loop. The command `Where(` *number* `)`, which replaces `Backtrace` of GAP 3, can be used to display *number* lines of information about the current function call stack.

As in GAP 3, access is only possible to the variables of the current level in the function stack, but in GAP 4 the function `DownEnv`, with a positive or negative integer as argument, permits one to step down or up in the stack.

When interrupting, the first line printed by `Where` actually may be one level higher, as the following example shows

```
gap> OnBreak := function() Where(0); end;; # eliminate back-tracing on
gap>                                       # entry to break loop
gap> test:= function( n )
>    if n > 3 then Error( "!\n" ); fi; test( n+1 ); end;;
gap> test( 1 );
Error, !
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> n;
4
brk> DownEnv();
brk> n;
3
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( 2 );
brk> n;
1
brk> Where();
 called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( -2 );
brk> n;
3
brk> quit;
gap> OnBreak := Where;; # restore OnBreak to its default value
```

For purposes of debugging, it can be helpful sometimes, to see what information is stored within an object. In GAP 3 this was possible using `RecFields` because the objects in question were represented via records. For component objects, GAP 4 permits the same by `NamesOfComponents(` *object* `)`, which will list all components present.