MiG Layout Cheat Sheet

Note! *Italics is used to denote an argument. Square brackets are used to indicate an optional argument.*

UnitValue A value that represents a size. Normally it consist of a value (integer or float) and the unit type (e.g. "`mm`"). MigLayout support defining custom unit types and there are some special ones built in. These are listed below and some have a context to which they can appear. UnitValues can be quite rich expressions, like: "`(10px + 0.25*((pref/2)-10))`".

The currently supported unit types are:

- **"" -** No unit specified. This is the default unit and pixels will be used by default. Default unit can be set with `PlatformDefaults.setDefaultHorizontal/VerticalUnit(int)`. E.g. "`10`"
- **px -** Pixels. Normal pixels mapped directly to the screen. E.g. "`10px`" or "`10`"
- **%** - A percentage of the container's size. May also be used for alignments where for instance `50%` means "centered". E.g. "`100%`"
- **lp -** Logical Pixels. If the normal font is used on the platform this maps `1:1` to pixels. If larger fonts are used the logical pixels gets proportionally larger. Used instead of Dialog Units. E.g. "`10lp`"
- **pt -** Points. 1/72:th of an inch. A unit normally used for printing. Will take the screen DPI that the component is showing on into account. E.g. "`10pt`"
- **mm -** Millimeters. Will take the screen DPI that the component is showing on into account. E.g. "`10mm`"
- **cm -** Centimeters. Will take the screen that the component is showing on DPI into account. E.g. "`10cm`"
- **in -** Inches. Will take the screen DPI that the component is showing on into account. E.g. "`10.4in`"
- **sp -** Percentage of the screen. Will take the pixel screen size that the component is showing on into account. `100.0` is the right/bottom edge of the screen. E.g. "`sp 70`" or "`sp 73.627123`"
- **al** - Visual bounds alignment. "`0al`" is left aligned, "`0.5al`" is centered and "`1al`" is right aligned. This unit is used with absolute positioning. E.g. "`0.2al`"
- **n/null** - Null value. Denotes the absence of a value. E.g. "`n`" or "`null`"

These are the unit values that are converted to pixels by the default `PlatformConverter`. The converted pixel sizes can be different for the vertical and horizontal dimension.

- **r/rel/related -** Indicates that two components or columns/rows are considered related. The exact pixel size is determined by the platform default. E.g. "`r`" or "`related`"
- **u/unrel/unrelatedated -** Indicates that two components or columns/rows are considered **un**related. The exact pixel size is determined by the platform default. E.g. "`u`" or "`unrelated`"
- **p/para/paragraph -** A spacing that is considered appropriate for a paragraph is used. The exact pixel size is determined by the platform default. E.g. "`para`" or "`paragraph`"
- **i/ind/indent -** A spacing that is considered appropriate for indent. The exact pixel size is determined by the platform default. E.g. "`i`" or "`indent`"

These are the unit values that can be specified as a reference to component(s) sizes. These can be used on column/row constraint's size and as a reference in component constaint expressions.

- **min/minimum -** A reference to the **largest** minimum size of the column/row. E.g. "`min`" or "`minimum`"
- **p/pref/preferred -** A reference to the **largest** preferred size of the column/row. E.g. "`p`" or "`pref`" or "`preferred`"
- **max/maximum -** A reference to the **smallest** maximum size of the column/row. E.g. "`max`" or "`maximum`"

These are the unit values that can be specified for a component's width. These can **only** be used on the **width** component constraints size.

- **button -** A reference to the platform minimum size for a button. E.g. "`wmin button`"

BoundSize A bound size is a size that optionally has a lower and/or upper bound and consists of one to three Unit Values. Practically it is a minimum/preferred/maximum size combination but none of the sizes are actually mandatory. If a size is missing (e.g. the preferred) it is **null** and will be replaced by the most appropriate value. For components this value is the corresponding size (E.g. `Component.getPreferredSize()` on Swing) and for columns/rows it is the size of the components in the row (see **min**/**pref**/**max** in **UnitValue** above).

The format is "*`min`*`:`*`preferred`*`:`*`max`*", however there are shorter versions since for instance it is seldom needed to specify the maximum size.

A single value (E.g. "`10`") sets only the *`preferred`* size and is exactly the same as "`null:10:null`" and "`:10:`" and "`n:10:n`".
Two values (E.g. "`10:20`") means minimum and preferred size and is exactly the same as "`10:20:null`" and "`10:20:`" and "`10:20:n`"
The use a of an exclamation mark (E.g. "`20!`") means that the value should be used for all size types and no colon may then be used in the string. It is the same as "`20:20:20`".

push can be appended to a gap to make that gap "greedy" and take any left over space. This means that a gap that has "`push`" will be pushing the components/rows/columns apart, taking as much space as possible for the gap. The gap push is always an addition to a **BoundSize**. E.g. "`gap rel:push`", "`[][]push[][]`", "`10cm!:push`" or "`10:10:10:push`".

Note! For row/column constraints the **minimum**, **preferred** and **maximum** keywords can be used and they refer to the largest minimum, preferred and maximum component in the column/row. A `null` value is the same thing as any of these constraints, for the indicated position, but they can for instance be used to set the minimum size to the preferred

one or the other way around. E.g. "pref:pref" or "min:min:pref".

AlignKeyword
For alignment purposes these keywords can be used: **t/top**, **l/left**, **b/bottom**, **r/right**, **lead/leading**, **trail/trailing** and **base/baseline**. Leading/trailing is dependant on if component orientation is "left-to-right" or "right-to-left". There is also a keyword "align label" or for columns/rows one need only to use "label". It will align the component(s), which is normally labels, left, center or right depending on the style guides for the platform. This currently means left justfied on all platforms except OS X which has right justified labels.

Layout Constraints

Layout constrainings and normally set in the constructor of MigLayout and is constraints that will affect the whole container.

wrap [*count*]	Sets auto-wrap mode for the layout. This means that the grid will wrap to a new column/row after a certain number of columns (for horizontal flow) or rows (for vertical flow). The number is either specified as an integer after the keyword or if not, the number of column/row constraints specified will be used. A wrapping layout means that after the count:th component has been added the layout will wrap and continue on the next row/column. If wrap is turned off (default) the Component Constraint's "wrap" and "newline" can be used to control wrapping.	"wrap" "wrap 4"
gap *gapx* [*gapy*] **gapx** *gap* **gapy** *gap*	Specifies the default gap between the cells in the grid and are thus overriding the platform default value. The gaps are specified as a **BoundSize**. See above.	"gap 5px 10px" "gap unrel rel" "gapx 10::50" "gapy 0:rel:null" "gap 10! 10!"
debug [*millis*]	Turns on debug painting for the container. This will lead to an active repaint every millis milliseconds. Default value is 1000 (once every second).	"debug" "debug 4000"
nogrid	Puts the layout in a flow-only mode. All components in the flow direction will be put in the same cell and will thus not be aligned with component in other rows/columns. For normal horizontal flow this is the same as to say that all component will be put in the first and only column.	"nogrid"
novisualpadding	Turns off padding of visual bounds (e.g. compensation for drop shadows)	"novisualpadding"
fill **fillx** **filly**	Claims all available space in the container for the columns and/or rows. At least one component need to have a "grow" constaint for it to fill the container. The space will be divided equal, though honoring "growpriority". If no columns/rows has "grow" set the grow weight of the componets in the rows/columns will migrate to that row/column.	"fill" "fillx" "filly"
ins/insets ["dialog"] ["panel"] [*top/all* [*left*] [*bottom*] [*right*]]	Specified the insets for the laid out container. The gaps before/after the first/last column/row overrides these layout insets. This is the same thing as setting an EmptyBorder on the container but without removing any border already there. Default value is "panel" (or zero if there are docking components). The size of "dialog" and "panel" insets is returned by the current PlatformConverter. The inset values all around can also be set explicitly for one or more sides. Insets on sides that are set to "null" or "n" will get the default values provided by the PlatformConverter. If less than four sides are specified the last value will be used for the remaining side. The gaps are specified as a **UnitValue**. See above. Note that the default insets is "panel"	"insets dialog" "ins 0" "insets 10px n n n" "insets 10 20 30 40"
flowy	Puts the layout in vertical flow mode. This means that the next cell is normally below and the next component will be put there instead of to the right. Default is horizontal flow.	"flowy"
al/align *alignx* [*aligny*] **aligny**/**ay** *align* **aligny**/**ax** *align*	Specifies the alignment for the laid out components as a group. If the total bounds of all laid out components does not fill the entire container the align value is used to position the components within the container without changing their relative positions. The alignment can be specified as a **UnitValue** or **AlignKeyword**. See above. If an **AlignKeyword** is used the "align" keyword can be omitted. Note that baseline alignment does not work since this is not for single components.	"align 50% 50%" "aligny top" "alignx leading" "align 100px" "top, left"
ltr/**lefttoright** **rtl**/**righttoleft**	Overrides the container's ComponentOrientation property for this layout. Normally this value is dependant on the Locale that the application is running. This constraint overrides that value.	"ltr" "lefttoright" "rtl"
ttb/**toptobottom** **btt**/**bottomtotop**	Specifies if the components should be added in the grid **bottom-to-top** or **top-to-bottom**. This value is not picked up from the container and is **top-to-bottom** by default.	"ttb" "toptobottom" "btt"
hidemode	Sets the default hide mode for the layout. This hide mode can be overridden by the component constraint. The hide mode specified how the layout manager should handle a component that isn't visible. The modes are: 0 - Default. Means that invisible components will be handled exactly as if they were visible. 1 - The size of an invisible component will be set to 0, 0. 2 - The size of an invisible component will be set to 0, 0 and the gaps will also be set to 0 around it. 3 - Invisible components will not participate in the layout at all and it will for instance not take up a grid cell.	"hidemode 1"
nocache	Instructs the layout engine to not use caches. This should normally only be needed if the "%" unit is used as it is a function of the parent size. If you are experiencing revalidation problems you can try to set this constraint.	"nocache"

Column/Row Constraints

Column and row constraints works the same and hence forth the term **row** will be used for both columns and rows.

Every [] section denotes constraints for that row. The gap size between is the gap size dividing the two rows. The format for the constraint is:
[constraint1, constraint2, ...]gap size[constraint1, constraint2, ...]gap size[...]..."

Example: "[fill]10[top,10:20]", "[fill]push[]", "[fill]10:10:100:push[top,10:20]".

Tip! A vertical bar "|" can be used instead of "][" between rows if the default gap should be used. E.g. "[100|200|300]" is the same as "[100][200][300]".

Gaps are expressed as a **BoundSize** (see above) and can thus have a min/preferred/max size. The size of the row is expressed the same way, as a **BoundSize**. Leaving any of the sizes out will make the size the default one. For gaps this is "related" (the pixel size for "related" is determined by the PlatformConverter) and for row size this is the largest of the contained components for minimum and preferred size and no maximum size. If there are fewer rows in the format string than there are in the grid cells in that dimension the last gap and row constraint will be used for the extra rows. For instance "[10]" is the same as "[10][10][10]" (affects wrapping if wrap is turned on though) .

Gaps have only their size, however there are number of constraints that can be used between the [ ] and they will affect that row.

":push" (or "push" if used with the default gap size) can be added to the gap size to make that gap greedy and try to take as much space as possible without making the layout bigger than the container.

Note! "" is the same as "[]" which is the same as "[pref]" and "[min:pref:n]".

sizegroup [*name*] **sg** [*name*]	Gives the row a size group name. All rows that share a size group name will get the same **BoundSize** as the row with the largest min/preferred size. This is most usable when the size of the row is not explicitly set and thus is determined by the largest component is the row(s). An empty name "" can be used unless there should be more than one group.	"sg" "sg group1" "sizegroup props"
fill	Set the *default value for components* to "grow" in the dimension of the row. So for columns the components in that column will default to a "growx" constraint (which can be overridden by the individual component constraints). Note that this property does not affect the size for the row, but rather the sizes of the components in the row.	"fill"
nogrid	Puts the row in flow-only mode. All components in the flow direction will be put in the same cell and will thus not be aligned with component in other rows/columns. This property will only be adhered to if the row is in the flow direction. So for the normal horizontal flow ("flowx") it is only used for rows and for "flowy" it is only used for columns.	"nogrid"
grow [*weight*]	Sets how keen the row should be to grow in relation to other rows. The weight (defaults to 100 if not specified) is purely a relative value to other rows' weight. Twice the weight will get double the extra space. If this constraint is not set, the grow weight is set to zero and the column will not grow (unless "fill" is set in the Layout Constraints and no other row has grow weight above zero either). Grow weight will only be compared to the weights for rows with the same grow priority. See below.	"grow 50" "grow"
growprio *prio*	Sets the grow priority for the row (not for the components in the row). When growing, all rows with higher priorities will be grown to their maximum size before any row with lower priority are considered. The default grow priority is 100. This can be used to make certain rows grow to max before other rows even start to grow.	"growprio 50"
shrink*weight*	Sets how keen/reluctant the row should be to shrink in relation to other rows. The weight is purely a relative value to other rows' weights. Twice the weight will shrink twice as much when space is scarce. If this constraint is not set the shrink weight defaults to 100, which means that all rows by default can shrink to their minimum size, but no less. Shrink weight will only be compared against the weights in the same shrink priority group (other rows with the same shrink priority). See below.	"shrink 50" "shrinkweight 0"
shrinkprio *prio* **shp** *prio*	Sets the shrink priority for the row (not for the components in the row). When space is scarce and rows needs to be shrunk, all rows with higher priorities will be shrunk to their minimum size before any row with lower priority are considered. The default shrink priority is 100. This can be used to make certain rows shrink to min before other rows even start to shrink.	"shrinkprio 50" "shp 110"
align *align* **al** *align*	Specifies the default alignment for the components in the row. This default alignment can be overridden by setting the alignment for the component in the Component Constraint. The default row alignment is "left" for columns and "center" for rows. The alignment can be specified as a **UnitValue** or **AlignKeyword**. See above. If **AlignKeyword** is used the "align" part can be omitted. Note that baseline alignement does not work if the component can't get its preferred size in the vertical dimension.	"align 50%" "align top" "al leading" "align 100px" "top, left" "align baseline"
gap *gapbefore* [*gap*] **gapbefore** *gap* **gapafter** *gap*	Specifies the gap before and/or after the row. The gap are specified between the row constraints (between "] ["). "gapleft", "gapright", "gaptop", "gapbottom" can also be used.	"gap 10 20" "gap 10:20:30 10px:20%:30in" "gapbefore 10px, gapafter 20px"

Component Constraints

Component constraints are used as an argument in the Container.add(...) for Swing and by setting it as Control.setLayoutData(...) in SWT. It can be used to specify constraints that has to do with the component's size and/or the grid cell flow. The constraints are specified one by one with comma signs as separators.
E.g. "width 100px!, grid 3 2, wrap".

wrap [*gapsize*]	Wraps to a new column/row **after** the component has been put in the next available cell. This means that the **next** component will be put on the new row/column. Tip! Read wrap as "wrap after". If specified "gapsize" will override the size of the gap between the current and next row (or column if "flowy"). Note that the gaps size is **after** the row that this component will end up at.	"wrap" "wrap 15px" "wrap push" "wrap 15:push"
newline [*gapsize*]	Wraps to a new column/row **before** the component is put in the next available cell. This means that the **this** component will be put on a new row/column. Tip! Read wrap as "on a newline". If specified "gapsize" will override the size of the gap between the current and next row (or column if "flowy"). Note that the gaps size is **before** the row that this component will end up at.	"newline" "newline 15px" "newline push" "newline 15:push"
push [*weightx*] [*weighty*] **pushx** [*weightx*] **pushy** [*weighty*]	Makes the row and/or column that the component is residing in grow with "weight". This can be used instead of having a "grow" keyword in the column/row constraints.	"push" "pushx 200" "pushy"
skip [*count*]	Skips a number of cells in the flow. This is used to jump over a number of cells before the next free cell is looked for. The skipping is done before this component is put in a cell and thus this cells is affected by it. "count" defaults to 1 if not specified.	"skip" "skip 3"
span [*countx*] [*county*] **spany**/**sy** [*count*] **spanx**/**sx** [*count*]	Spans the current cell (merges) over a number of cells. Practically this means that this cell and the *count* number of cells will be treated as one cell and the component can use the space that all these cells have. *count* defaults to a really high value which practically means *span to the end or the row/column*. Note that a cell can be spanned **and** split at the same time, so it can for instance be spanning 2 cells and split that space for three components. "span" for the first cell in a row is the same thing as setting "nogrid" in the row constraint.	"span" "span 4" "span 2 2" "spanx 10" "spanx 2, spany 2"
split [*count*]	Splits the cell in a number of sub cells. Basically this means that the next count number of components will be put in the same cell, next to each other with defait gaps. Only the first component in a cell can set the split, any subsequent "split" keywords in the cell will be ignored. count defaults to *infinite* if not specified, which means that "split" alone	"split" "split 4"

	will put all subsequent components in the same cell. "skip", "wrap" and "newline" will break out of the split cell. The latter two will move to a new row/column as usual. "skip" will skip out if the splitting and continue in the next cell.	
cell *col row* [span x [span y]]	Sets the grid cell that the component should be placed in. If there are already components in the cell they will share the cell. If there are two integers specified they will be interpreted as absolute coordinates for the column and row. The flow will continue after this cell. How many cells that will be spanned is optional but may be specified. It is the same thing as using the spanx and spany keywords.	"cell 2 2" "cell 1 1 2 2"
flowx **flowy**	Sets the flow direction in the cell. By default the flow direction in the cell is the same as the flow direction for the layout. So if the components flows from left to right they will do so for in-cell flow as well. The first component added to a cell can change the cell flow. If flow direction is changed to flowy the components in the cell will be positioned above/under each other.	"flowy" "flowx"
w/width *size* **h/height** *size*	Overrides the default size of the component that is set by the UI delegate or by the developer explicitly on the component. The size is specified as a **BoundSize**. See the *Common Argument Types* section above for an explanation. Note that expressions is supported and you can for instance set the size for a component with "width pref+10px" to make it 10 pixels larger than normal or "width max(100, 10%)" to make it 10% of the container's width, but a maximum of 100 pixels.	"width 10!" "width 10" "h 10:20" "height pref!" "w min:100:pref" "w100!,h100!" "width visual.x2-pref"
wmin/wmax *x-size* **hmin/hmax** *y-size*	Overrides the default size of the component for minimum or maximum size that is set by the UI delegate or by the developer explicitly on the component. The size is specified as a **BoundSize**. See the *Common Argument Types* section above for an explanation. Note that expressions is supported and you can for instance set the size for a component with "wmin pref-10px" to make it no less than 10 pixels smaller than normal. These keywords are syntactic shorts for "width *size*:pref" or "width min:pref:*size*" with is exactly the same for minimum and maximum respectively.	"wmin 10" "hmax pref+100"
grow [*weightx*] [*weighty*] **growx** [*weightx*] **growy** [*weighty*]	Sets how keen the component should be to grow in relation to other component in the same cell. The weight (defaults to 100 if not specified) is purely a relative value to other components' weight. Twice the weight will get double the extra space. If this constraint is not set the grow weight is set to 0 and the component will not grow (unless fill is set in the row/column in which case "grow 0" can be used to explicitly make it not grow). Grow weight will only be compared against the weights in the same grow priority group and for the same cell. See below.	"grow 50 20" "growx 50" "grow" "growx" "growy 0"
growprio/**gp** *prio* **prowpriox**/**gpx** *prio* **growprioy**/**gpy** *prio*	Sets the grow priority for the component. When growing, all components with higher priorities will be grown to their maximum size before any component with lower priority are considered. The default grow priority is 100. This constraint can be used to make certain components grow to max before other components even start to grow.	"growprio 50 50" "gp 110 90" "gpx 200" "growpriox 200"
shrink *weightx [weighty]*	Sets how keen/reluctant the component should be to shrink in relation to other components. The weight is purely a relative value to other components' weight. Twice the weight will shrink twice as much when space is scarce. If this constraint is not set the shrink weight defaults to 100, which means that all components by default can shrink to their minimum size, but no less. Shrink weight will only be compared against the weights in the same shrink priority group (other components with the same shrink priority). See below.	"shrink 50" "shrink 50 50"
shrinkprio/**shp** *priox [prioy]* **shrinkpriox**/**shpx** *priox* **shrinkprioy**/**shpy** *prioy*	Sets the shrink priority for the component. When space is scarce and components needs be be shrunk, all components with higher priorities will be shrunk to their minimum size before any component with lower priority are considered. The default shrink priority is 100. This can be used to make certain components shrink to min before other even start to shrink.	"shrinkprio 50" "shp 200 200" "shpx 110"
sizegroup/**sg** [*name*] **sizegroupx**/**sgx** [*name*] **sizegroupy**/**sgy** [*name*]	Gives the component a size group name. All components that share a size group name will get the same **BoundSize** (min/preferred/max). It is used to make sure that all components in the same size group gets the same min/preferred/max size which is that of the largest component in the group. An empty name "" can be used.	"sg" "sg group1" "sizegroup props" "sgx" "sizegroupy grp1"
endgroup/**eg** [*name*] **endgroupx**/**egx** [*name*] **endgroupy**/**egy** [*name*]	Gives the component an end group name and association. All components that share an end group name will get their right/bottom component side aligned. The right/bottom side will be that of the largest component in the group. If "eg" or "endgroup" is used and thus the dimension is not specified the current flow dimension will be used (see "flowx"). So "eg" will be the same as "egx" in the normal case. An empty name "" can be used.	"eg" "eg group1" "endgroup props" "egx" "endgroupy grp1"
gap *left* [*right*] [*top*] [*bottom*] **gaptop** *gap* **gapleft** *gap* **gapbottom** *gap* **gapright** *gap* **gapbefore** *gap* **gapafter** *gap*	Specifies the gap between the components in the cell or to the cell edge depending on what is around this component. If a gap size is missing it is interpreted as 0px. The gaps are specified as a **BoundSize**. See above.	"gap 5px 10px" "gap unrel rel" "gapx 10:20:50" "gapy 0:rel:null" "gap 10! 10!"
gapx *left* [*right*] **gapy** *top [bottom]*	Specifies the horizontal or vertical gap between the components in the cell or to the cell edge depending on what is around this component. If a gap size is missing it is interpreted as 0px. The gaps are specified as a **BoundSize**. See above.	"gapx 5px 10px" "gapy unrel rel"
id [*groupid.*] id	Sets the id (or name) for the component. If the id is not specified the ComponentWrapper.getLinkId() value is used. This value will give the component a way to be referenced from other components. Two or more components may share the group id but the id should be unique within a layout. The value will be converted to lower case and are thus **not** case sensitive. There must not be a dot first or last in the value string.	"id button1" "id grp1.b1"

pos *x y [x2] [y2]*	Positions the component with absolute coordinates relative to the container. If this keyword is used the component will **not** be put in a grid cell and will thus not affect the flow in the grid. One of either x/x2 and one of y/y2 must not be `null`. The coordinate that is set to `null` will be placed so that the component get its preferred size in that dimension. Non-specified values will be set to `null`, so for instance "`abs 50% 50%`" is the same as "`abs 50% 50% null null`". If the position and size can be determined without references to the parent containers size it will affect the preferred size of the container. Example: "`pos 50% 50% n n`" or "`pos 0.5al 0.5al`" or "`pos 100px 200px`" or "`position n n 200 200`". Absolute positions can also links to other components' bounds using their `id`s or `groupId`s. It can even use expressions around these links. E.g. "`pos (butt.x+indent) butt1.y2`" will position the component directly under the component with id "butt1", indented slightly to the right. There are two special bounds that are always set. "`container`" are set to the bounds if the container and "`visual`" are set to the bounds of the container minus the specified insets. The coordinates that can be used for these links are: • **.x** or **.y** - The top left coordinate of the referenced component's bounds • **.x2** or .**y2** - The lower right coordinate of the referenced component's bounds • **.w** or **.h** - The current width and height of the referenced component. • **.xpos** or **.ypos** - The top left coordinate of the referenced component **in screen coordinates**.	"`pos (b1.x+b1.w/2) (b1.y2+rel)`" "`pos (visual.x2-pref) 200`" "`pos n b1.y b1.x-rel b1.y2`" "`pos 100 100 200 200`"
x x **x2** x2 **y** y **y2** y2	Used to position the start (x or y), end (x2 or y2) or both edges of a component in absolute coordinates. This is used for when a component is in a grid or dock and it for instance needs to be adjusted to align with something else or in some other way be positioned absolutely. The cell that the component is positioned in will not change size, neither will the grid. The x, y, x2 and y2 keywords are applied in the last stage and will therefore not affect other components in the grid or dock, unless they are explicitly linked to the bounds of the component. If the position and size can be determined without references to the parent containers size it will affect the preferred size of the container.	"`x button1.x`" "`x2 (visual.x2-50)1`" "`x 100, y 300`"
dock ("*north*" "*west*" "*south*" "*east*") or **north**/**west**/**south**/**east**	Used for docking the component at an edge, or the center, of the container. Works much like `BorderLayout` except that there can be an arbitrary number of docking components. They get the docked space in the order they are added to the container and "cuts that piece of". The "`dock`" keyword can be omitted for all but "*center*" and is only there to use for clarity. The component will be put in special surrounding cells that spans the rest of the rows which means that the docking constraint can be combined with many other constraints such as `padding`, `width`, `height` and `gap`.	"`dock north`" "`north`" "`west, gap 5`"
pad *top [left] [bottom] [right]*	Sets the padding for the component in absolute pixels. This is an absolute adjustment of the bounds if the component and is done at the last stage in the layout process. This means it will not affect gaps or cell size or move other components. It can be used to compensate for something that for some reason is hard to do with the other constraints. For instance "`ins -5 -5 5 5`" will enlarge the component five pixels in all directions making it 10 pixels taller and wider. If values are omitted they will be set to 0. **Note!** Padding multi-line components derived from `JTextComponent` (such as `JTextArea`) without setting a explicit minimum size may result in an continuous size escalation (animated!). This is not a bug in the layout manager but a "feature" derived from how these components calculates their minimum size. If the size is padded so that it increases by one pixel, the text component will automatically issue a revalidation and the layout cycle will restart, now with a the newly increased size as the new **minimum** size. This will continue until the maximum size is reached. This only happens for components that have "line wrap" set to `true`.	"`padding 10 10`" "`pad 5 5 -5 -5`" "`pad 0 0 1 1`"
al/**align** *alignx* [*aligny*] **alignx**/**ax** *alignx* **aligny**/**ay** *aligny*	Specifies the alignment for the component if the cell is larger than the component plus its gaps. The alignment can be specified as a **UnitValue** or **AlignKeyword**. See above. If **AlignKeyword** is used the "`align`" keyword can be omitted. In a cell where there is more than one component, the first component can set the alignment for all the components. It is not possible to for instance set the first component to be left aligned and the second to be right aligned and thus get a gap between them. That effect can better be accomplished by setting a gap between the components that have a minimum size and a large preferred size. Note that baseline alignement does not work if the component can't get its preferred size in the vertical dimension.	"`align 50% 50%`" "`aligny top`" "`alignx leading`" "`align 100px`" "`top, left`" "`aligny baseline`"
external	Inhibits MigLayout to change the bounds for the component. The bounds should be handled/set from code outside this layout manager by calling the `setBounds(..)` (or equivalent depending on the GUI toolkit used) directly on the component. This component's bounds can still be linked to by other components if it has an "`id`" tag, or a link id is provided by the `ComponentWrapper`. This is a very simple and powerful way to extend the usages for MigLayout and reduce the number of times a custom layout manager has to be written. Normal application code can be used to set the bounds, something that can't be done with any other layout managers.	"`external`" "`external,id butt`"
hidemode	Sets the hide mode for the component. If the hide mode has been specified in the This hide mode can be overridden by the component constraint. The hide mode specified how the layout manager should handle a component that isn't visible. The modes are: `0` - Default. Means that invisible components will be handled exactly as if they were visible. `1` - The size of the component (if invisible) will be set to `0, 0`. `2` - The size of the component (if invisible) will be set to `0, 0` and the gaps will also be set to `0` around it. `3` - Invisible components will not participate in the layout at all and it will for instance not take up a grid cell.	"`hidemode 1`"
tag [*name*]	Tags the component with metadata name that can be used by the layout engine. The tag can be used to explain for the layout manager what the components is showing, such as an **OK** or **Cancel** button. Unknown tags will be disregarded without error or any indication. Currently the recognized tags are used for button reordering on a per platform basis. See the JavaDoc for `PlatformConverter.getButtonBarOrder(int type)` for a longer explanation. The supported tags are: • **ok** - An OK button. • **cancel** - A Cancel button. • **help** - Help button that is normally on the right. • **help2** - Help button that on some platforms is placed to the left.	"`tag ok`" "`tag help2`"

	• **yes -** A Yes button. • **no -** A No button. • **apply -** An Apply button. • **next -** A Next or Forward button. • **back -** A Previous or Back button. • **finish -** A Finished button. • **left -** A button that should normally always be placed on the far left. • **right -** A button that should normally always be placed on the far right.	