MigLayout

MigLayout is an extremely flexible layout manager that can be used in many different ways to layout components using a single and consistent API.

The main features are:

- String based constraints makes the layout code portable, short, human readable and easy to implement for XUL and GUI builders.
- From v2.0 there is also an API to create constraints. This means that whatever your preference is, String or API, you can use Mig Layout to create your layouts.
- MiG Layout is the most flexible layout engine to date, capable of doing everything that **all** major Layout Managers can, with one API.
- Extensible Units that can can be used to layout components with everything from screen percentage to millimeters.
- Docking, absolute positioning with component links and powerful grid layout all in one layout manager.
- GUI toolkit dependence put in three simple wrapper classes makes MiG Layout trivial to port to for instance .NET or any other GUI toolkit.
- MiG Layout is free to use and is Open Source.

Initial Example

An initial example that uses the grid funcitonality to create two rows with a right aligned label and a growing text field on each of the rows. It is using the default ("related") gaps except for the inter-row gap which is 10 pixels.

```
// Layout, Column and Row constraints as arguments.
MigLayout layout = new MigLayout("fillx", "[right]rel[grow,fill]", "[]10[]");
JPanel panel = new JPanel(layout);

panel.add(new JLabel("Enter size:"),   "");
panel.add(new JTextField(""),          "wrap");
panel.add(new JLabel("Enter weight:"), "");
panel.add(new JTextField(""),          "");
```

Or the same layout with the API constraint building. Sice they are so similar the API version will not be handled much further in this white paper.

```
// Layout, Column and Row constraints as arguments.
MigLayout layout = new MigLayout(
     new LC().fillX(),
         new AC().align("right").gap("rel").grow().fill(),
         new AC().gap("10");

JPanel panel = new JPanel(layout);

panel.add(new JLabel("Enter size:"));
panel.add(new JTextField(""),            new CC().wrap());
panel.add(new JLabel("Enter weight:"));
```

```
panel.add(new JTextField(""));
```

Introduction

The constraints used are all entered as Strings or through chained API method calls. The API and String constraints are the same with some small differences that are documented in the JavaDoc for the API methods. The API version will not be handleled furthere here since that would mostly be reiterating the same things twice. You can download the JavaDoc on the Mig Layout site.

There are three major ways to layout components with MigLayout. They can be combined freely in the same container. You can for instance link a component's position to a component in the grid with absolute positioning and have it use the same vertical size as another component that is docked "north" in the container. These are the main layout types:

- **Grid Based**. This is the default mode and what is happening if you just add components and don't specify any of the other ways. It is more flexible than both GridBagLayout and JGoodies' FormLayout.
- **Docking Components**. This add components at any of the four edges of the container, or in the center. It is more flexible and than BorderLayout yet as easy to use.
- **Absolute positioning with Links**. Components can be positioned with absolute coordinates and linked to other components' bounds, to the container, to component groups' bounds or to any combination of this using normal Java expressions.

MigLayout is using a grid (rows and columns) with automatic handling of gaps for default basic component layout. The grid is very flexible and can for instance be tweaked more that an HTML table. For instance, every cell can be split to contain more than one component, and several cells can be spanned (merged) to work as one big cell. The cells can even both be spanned and split at the same time making almost any conceivable layout possible, without resorting to "tricks".

The components in a cell, if more than one, will be flowed much like FlowLayout do, but with more control over how it's done and without involuntary wrapping. Flow can be vertical (y) or horizontal (x) and both the main grid and the individial cells can have different flow directions, though they default to horizontal flow.

MigLayout can also position components absolutely or relative to the container's bounds, or even *visual* bounds, which is the bounds minus the insets. It does this outside the bounds of a grid. One can even link components to eachothers' bounds.

MigLayout can use many different unit types, such as millimeters or inches. It also has numerous features to create stable and good looking layouts, many of which is known from FormLayout and GroupLayout.

Do not let the vast amount of options make you think that this layout manager is hard to use. They are seldom needed for normal layouts but they are very handy to have when exact and complex layouts are required.

Basic Concepts

There are three constraint types that can be set on the layout manager instance and the handled components. They are:

- **Layout Constraints**. These constraints specify how the layout manager instance should work in general. For instance how all the laid out components should be aligned as a group, should

there be available space in the container. This constraint is set directly on the layout manager instance, either in the constructor or using a standard get/set property. E.g. `"align center, fill"`.

- **Row/Column Constraints**. Specifies the properties for the grid's rows and columns. Constraints such as sizes and default alignments can be specified. These constraints are set directly on the layout manager instance, either in the constructor or using standard get/set properties. E.g. `"[35px]10px[50:pref]"`.
- **Component Constraints**. They are used for specifying the properties and bounds for individual components. This can for instance be to override the minimum, preferred or maximum vertical and/or horizontal sizes for the component. You can set the alignment and if a cell should be split and/or spanned, and much more. This constraint is normally set as an argument when adding the component to the container or using standard get/set properties. E.g. `"width 100px, left"`.

Grid Flow

The components are normally laid out in a grid. Which components goes into which cells are decided in two ways, or by a mix thereof. When the first component is added to the container the MigLayout that is managing it will put the component in the (0, 0) cell. The next one will be put in the (1, 0) and so on. This is the normal flow, however you can change the default direction of the flow in any way. See the Layout Constraints below for what you can do and how to do it. You can also set the cell by providing a column/row coordinate in the Component Constraint E.g. `"grid 2 4"`. That component will end up in that cell and the next component will flow from there the same way as described above, unless it also has an absolute grid position specified. There are also flow constraints such as `"wrap"`, `"newline"`, `"skip"`, `"span"` and `"split"` to control the flow in the grid, like how normal text flows over a paper or computer screen.

In-cell Flow

If more that one component is occupying the same cell (either because the cell has been split or the same cell has been specified in the Component Constraints by a grid position) the components in that cell will flow in the vertical or horizontal direction. Default direction will be the same as the whole layout is using, but it can be set on a cell-by-cell basis. Components will be put next to each other with an option to align them in the non-flowing direction (e.g. `"aligny"` for horizontal flow). The gaps between the components have customizable min/preferred/max sizes and this can for instance be used to create glues (pushing gaps).

Gaps

The white space between components is highly configurable. The gap between each cell row/column can be specified on a row/column basis. The default gap is decided per platform and defaults to the interpretation of how mush a `"related"` gap is. The default gap can be overridden for the whole layout in the Layout Constraint or it can be specified in the row/column constraints. Generally gaps are not added; instead they are considered to be the minimum space a component/cell wants to the components/cells around, this is consistent with how humans would define spacing. If for instance one component have a gap of 10 pixels after it and the next component has specified a 20 pixel gap before it, the resulting gap is **20** pixels and not 30. Gaps can actually be specified in a min/preferred/max size notion which gives them power to for instance grow/shrink as available space changes. Gaps between columns/rows and/or components in a cell can be appended with ":push" wich means that the gap should be greedy and use any free space avaible. Pushing gaps in a cell will not force the cell to be bigger, it will only use the left over space in the cell. Pushing gaps between columns/rows will make the layout fill the whole space of the container, if any is avalable.

Absolute Positioning

In the Component Constraints it is possible to specify absolute coordinates in any unit and the component will get these and thus **not** be positioned in the grid cells. These coordinates will be relative to the parent so it is for instance easy to put a component and 20% in and 80% down into the container. This is a powerful and flexible way to layout components that needs to be independent of other components and possibly overlap them. For instance this can be used to create a local `GlassPane`-like component over a container, without the need for re-routing events and such tricks needed for normal `GlassPanes`.

All coordinates in the absolute constraint (I.e. x, y, x2, y2) can have links to other components or to the parent, or its visual bounds. Since these coordinates are `UnitValues` they can in fact be full expressions. For inststance `(b1.x + indent)` as an x-coordinate will put the left side of the component indented relative to `"b1"`.

Expressions

Everywhere where there is a `UnitValue` there can always be a full expression that looks like a normal Java expression. For instance `(button1.x + 1cm - (parent.w*0.2))` will be evaluated the same way it would in Java code, with precedence for multiply and division and full support for parentheses. If spaces is used in the expression it must be surrounded by parentheses.

Docking Components

Docking is fully supported by MigLayout. The docking is similar to the docking in `BorderLayout` but it is much more configurable and can have more than five components docked. Components can be glued to one of the edges: `"north"`, `"west"`, `"south"`,`"east"` or `"center"` and will "cut that part off" so that no other components will be laid out there (except maybe absolute positioned components). The space that is left in the middle (center) is laid out with the normal grid, the normal way, unless there is a `"dock center"` component. Docking is a very flexible way to layout panels but it can be used for many other layout needs as well.

Button Bars and Button Order

MigLayout has support for reorganizing your buttons (within a single cell, which might span a whole row though) according to platform preference. For instance the **OK** and **Cancel** buttons have different order on **Windows** and **Mac OS X**. While other layout managers use factories and button builders for this, it is inherently supported by MigLayout by just tagging the buttons. One just tags the **OK** button with `"ok"` and the **Cancel** button with `"cancel"` and they will end up in the correct order for the platform the application is running on, if they are put in the same grid cell. There are about 10 different tags and the format for how the buttons should be ordered, and what spacing is to be used between buttons, including glues, can be highly customized in a very simple way by settting the button order with `PlatformDefaults.setButtonOrder(String)`.

Different Units

There is support for a lot of different units in MigLayout. The built in units are **Pixel**, **Millimeter**, **Centimeter**, **Points**, **Inch**, **LogicalPixel**, **Percent** and **ScreenPercent**. There are also a few special keywords that can be used, such as `"related"`, `"unrelated"`, `"indent"` and `"paragraph"`. They are used to make gaps (spacing) appropriate to whatever platform the application is running on and they are a great help to make the applications feel native on every platform.
You can even create your own units by providing a converter between that unit and pixels.

Growing and Shrinking

What should happen when a component isn't given the preferred size is extremely customizable. There is the option to divide components into grow and shrink priority groups where a higher priority will shrink/grow before components/rows/columns with lower priorities are resized. Grow weight

within those groups can be set to specify how the free space or shrinkage should be divided between the components/rows/columns in *that priority group*. There are a lot of things to configure if needed, but the normal behavior is enough for most cases. MigLayout fully adheres to the Component's minimum, preferred and maximum sizes.

Size Groups

Sometimes there is the need for components/columns/rows to have the same width and/or height. This is easily done with *size groups*. These can be used for instance in button bars where buttons normally should have the same minimum width or to create non-fluctuating component form in tab panes.

End Groups

For absolute layouts there is sometimes the need to set the end of related components to the biggest one in the group. This can be accomplished in an easy way by assigning the components an `endgroup`. All components in the same end group will automatically have their right side set to the largest component in that group.

Component Sizes

The components have minimum, preferred and maximum size which are normally set by the UI delegate behind the component or explicitly by the component user. In the Component Constraints these sizes can be overridden so that the need for explicitly setting the sizes directly on the component is seldom needed. If not overridden the original values will be used. This is even more useful in GUI toolkits that doens't have the notion of minimum and maximum component sizes, such as Eclipse's SWT Component Toolkit. With MigLayout you can set minimum and/or maximum size in the component constraints and this will work the same way as if they were set directly on the component, as you might do in for instance Swing.

Visual Bounds

Some components, for instance the `JTabbedPane` in the Windows XP theme, have drop shadows. This means that the visual, human perceptible, bounds are not the same as the mathematical outer bounds. MigLayout has a way to handle this and it can be extended to include information for more components in the future.

Baseline Support

MigLayout supports baseline alignment for **JDK 6** and later. It does this using reflection so that releases before **JDK 6** can be used. Baseline defaults to center alignment if not supported by the JDK. Components in the same grid row can be baseline aligned. Default alignment can be set to baseline, or it can be specified for individual components.

Visual Guide Lines for Debugging Layouts

To turn on debugging for a container just add the `"debug"` constraint to the Layout Constraints. This will trigger a special debug overdraw of the container that will outline the cells in the grid and the component bounds. This is a invaluable help for understanding why components end up where they do. There is no need for a DebugPanel or anything else that will modify the actual component creation code.

Common Argument Types

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.setDefaultUnit(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. E.g. `"10lp"`
- **pt** - Points. 1/72:th of an inch. 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. `"10 mm"`
- **cm** - Centimeters. Will take the screen that the component is showing on DPI into account. E.g. `"10 cm"`
- **in** - Inches. Will take the screen DPI that the component is showing on into account. E.g. `"10in"`
- **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"`

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"`

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"`

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. 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. 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

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.
Example: `"wrap" or "wrap 4"`.

gap *gapx* [*gapy*] or **gapx** *gapx* or **gapy** *gapy*
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.
Example: "gap 5px 10px" or "gap unrel rel" or "gapx 5dlu" or "gapx 10::50" or "gapy 0:rel:null" or "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).
Example: "debug" or "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.
Example: "nogrid".

novisualpadding
Turns off padding of visual bounds (e.g. compensation for drop shadows)
Example: "novisualpadding".

fill or **fillx** or **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 component in the rows/columns will migrate to that row/column.
Example: "fill" or "fillx" or "filly"

ins/insets ["dialog"] or ["panel"] or [*top/all* [*left*] [*bottom*] [*right*]]
Specified the insets for the laid out container. The gaps before/after the first/last column/row is always discarded and replaced by 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.
Example: "insets dialog" or "ins 0" or "insets 10 n n n" or "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.
Example: "flowy"

al/align *alignx* [*aligny*] or **aligny**/**ay** *aligny* or **aligny**/**ax** *alignx*
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.
Example: "align 50% 50%" or "aligny top" or "alignx leading" or "align 100px" or "top, left" or "aligny baseline"

ltr/lefttoright or 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.
Example: `"ltr" or "lefttoright" or "rtl"`

ttb/toptobottom or 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.
Example: `"ttb" or "toptobottom" or "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.
Example: `"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.
Example: `"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]"`.

sg/**sizegroup** [*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.
Example: `"sg" or "sg grp1" or "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 size of the components in the row.
Example: `"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.
Example: `"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.
Example: `"grow 50" or "grow"`

growprio *priority*
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.
Example: `"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.
Example: `"shrink 50"`

shp/shrinkprio *priority*
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.
Example: `"shrinkprio 50" or "shp 110"`

al/align *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.
Example: `"align 50%" or "align top" or "align leading" or "align 100px" or "top, left" or "align baseline"`

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 format is same as for Layout Constraint, which means that the constraints are specified one by one with comma signs as separators.
Example: `"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.
Example: `"wrap" or "wrap 15px" or "wrap push" or "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.
Example: `"newline" or "newline 15px" or "newline push" or "newline 15:push"`

push [*weightx*] [*weighty*] or **pushx**[*weightx*] or **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.
Example: `"push" or "pushx 200"`.

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.
Example: "skip" or "skip 3".

span [*countx*] [*county*] or **spany**/**sy** [*count*] or **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.
Example: "span" or "span 4" or "span 2 2" or "spanx" or "spanx 10" or "spanx 2, spany 2".

split [*count*]
Splits the cell in a number of sub-cells. Practically this means that the next count number components will be put in the same cell, next to eachother without 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 will put all coming components in the same cell. "split", "wrap" or "newline" will break out of the split celll. The latter two will move to a new row/column as usual. Note! "skip" will will skip out if the splitting and continue in the next cell.
Example: "split" or "split 4".

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.
E.g. "cell 2 2" or "cell 0 1 2 " or "cell 1 1 3 3".

flowx or **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.
Example: "flowy" or "flowx".

w/width *size* or **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.
Example: "width 10!" or "width 10" or "h 10:20" or "height pref!" or "w min:100:pref" or "w 100!,h 100!" or "width visual.x2-pref".

wmin/wmax *x-size* or **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 maximun respectively.
Example: "wmin 10" or "hmax pref+100".

grow [*weightx*] [*weighty*] or **growx** [*weightx*] or **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.
Example: "grow 50 20" or "growx 50" or "grow" or "growx" or "growy 0"

growprio/**gp** *prio* or **growpriox**/**gpx** *priox* or **growprioy**/**gpy** *prioy*
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.
Example: "growprio 50 50" or "gp 110 90" or "gpx 200" or "prowprioy 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.
Example: "shrink 50" or "shrink 50 40"

shrinkprio/**shp** *priox [prioy]* or **shrinkpriox**/**shpx** *priox* or **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.
Example: "shrinkpriority 50 50 " or "shp 200 200 " or "shpx 110"

sizegroup/**sg** [*name*] or **sizegroupx**/**sgx** [*name*] or **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.
Example: "sg" or "sg group1" or "sizegroup props" or "sgx" or "sizegroupy grp1".

eg/endgroup [*name*] or **egx/endgroupx** [*name*] or **egy/endgroupy** [*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.

Example: `"eg" or "eg group1" or "endgroup props" or "egx" or "endgroupy grp1"`.

gap *left* [*right*] [*top*] [*bottom*] or **gaptop** *gap* or **gapleft** *gap* or **gapbottom** *gap* or **gapright** *gap* or **gapbefore** *gap* or **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.

Example: `"gap 5px 10px 5px 7px" or "gap unrel rel" or "gapx 5dlu" or "gapx 10:20:50:push" or "gapy 0:rel:null" or "gap 10! 10!" or "gapafter push"`.

gapx *left* [*right*] or **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.

Example: `"gapx 5px 10px" or "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.

Example: `"id button1" or "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**.

Example: `"pos (b1.x+b1.w/2) (b1.y2+rel)" or "pos (visual.x2 - pref) 200" or "pos n b1.y b1.x-rel b1.y2" "pos 100 100 200 200"`.

x *x* or **x2** *x2* or **y** *y* or **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.
Example: `"x button1.x" or "x2 (visual.x2-50)" or "x 100, y 300"`.

dock (`"north"` or `"west"` or `"south"` or `"east"` or `"center"`) 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`.
Example: `"dock north" or "north" or "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`.
Example: `"padding 10 10" or "pad 5 5 -5 -5" or "pad 0 0 1 1"`.

al/align *alignx* [*aligny*] or **aligny**/**ay** *aligny* or **aligny**/**ax** *alignx*

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.
Example: `"align 50% 50%" or "aligny top" or "alignx leading" or "align 100px" or "top, left" or "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.
Example: `"external"` or `"external, id mybutt"`.

`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.
Example: `"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.
- **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.

Example: `"tag ok" or "tag help2"`.

Further Reading

There are more information to digest regarding MigLayout. The resources are all available at www.migcomponents.com

- The demonstration application for Swing and SWT is available both as binaries and with free (BSD) source code. There might be other frameworks added in the future.
- There is a cheat sheet that lists the constraints in a short and simple layout and can be view in a browser (HTML) or printed (PDF).
- There is an online forum dedicated to MigLayout. The it can be found here: www.miginfocom.com/forum