His version of the example looks like this:

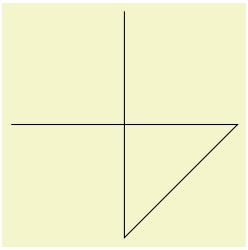
```
%% ConTeXt file
\usemodule[tikz]

\starttext
  We are working on
  \starttikzpicture
    \draw (-1.5,0) -- (1.5,0);
    \draw (0,-1.5) -- (0,1.5);
  \stoptikzpicture.
\stoptext
```

Hans will now typeset this file in the usual way using `texexec` or `context`.

2.3 Straight Path Construction

The basic building block of all pictures in Ti*k*Z is the path. A *path* is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in `(0,0)`. This is followed by a series of "path extension operations". The simplest is `--`, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



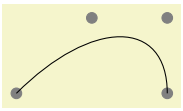
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl is a bit confused by the fact that there is no `{tikzpicture}` environment, here. Instead, the little command `\tikz` is used. This command either takes one argument (starting with an opening brace as in `\tikz{\draw (0,0) -- (1.5,0)}`, which yields ——————) or collects everything up to the next semicolon and puts it inside a `{tikzpicture}` environment. As a rule of thumb, all Ti*k*Z graphic drawing commands must occur as an argument of `\tikz` or inside a `{tikzpicture}` environment. Fortunately, the command `\draw` will only be defined inside this environment, so there is little chance that you will accidentally do something wrong here.

2.4 Curved Path Construction

The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, Ti*k*Z provides a special syntax. One or two "control points" are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y. Then the curve will start "going in the direction of y at x", that is, the tangent of the curve at x will point toward y. Next, suppose the curve should end at z and the second support point is w. Then the curve will, indeed, end at z and the tangent of the curve at point z will go through w.

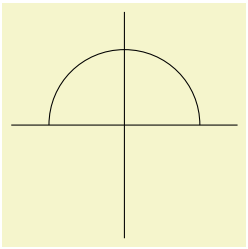
Here is an example (the control points have been added for clarity):



```
\begin{tikzpicture}
  \filldraw [gray] (0,0) circle [radius=2pt]
                   (1,1) circle [radius=2pt]
                   (2,1) circle [radius=2pt]
                   (2,0) circle [radius=2pt];
  \draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

The general syntax for extending a path in a "curved" way is `.. controls` ⟨*first control point*⟩ `and` ⟨*second control point*⟩ `..` ⟨*end point*⟩. You can leave out the `and` ⟨*second control point*⟩, which causes the first one to be used twice.

So, Karl can now add the first half circle to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

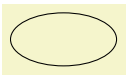
Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

2.5 Circle Path Construction

In order to draw a circle, the path construction operation `circle` can be used. This operation is followed by a radius in brackets as in the following example: (Note that the previous position is used as the *center* of the circle.)

```
\tikz \draw (0,0) circle [radius=10pt];
```

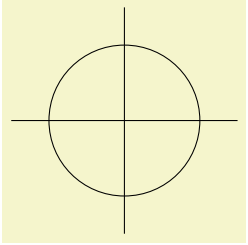
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them:



```
\tikz \draw (0,0) ellipse [x radius=20pt, y radius=10pt];
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction (a "turned ellipse" like ⬭) you can use transformations, which are explained later. The code for the little ellipse is `\tikz \draw[rotate=30] (0,0) ellipse [x radius=6pt, y radius=3pt];`, by the way.

So, returning to Karl's problem, he can write `\draw (0,0) circle [radius=1cm];` to draw the circle:

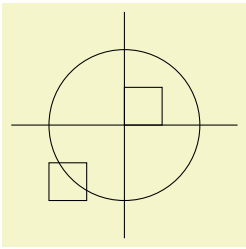


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that Ti*k*Z has powerful transformation options and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

2.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to produce it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the `rectangle` path construction operation. This operation should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (0,0) rectangle (0.5,0.5);
  \draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

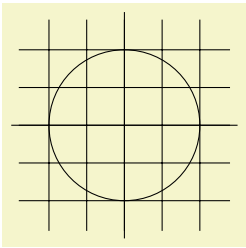
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `\draw` command, when he learns that there is a `grid` path construction operation.

2.7 Grid Path Construction

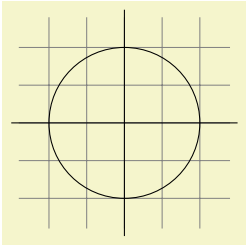
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces  . Note how the optional argument for `\draw` can be used to specify a grid width (there are also `xstep` and `ystep` to define the steppings independently). As Karl will learn soon, there are *lots* of things that can be influenced using such options.

For Karl, the following code could be used:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the `\draw` command that draws the grid. First, he uses the color `gray` for the grid lines. Second, he reduces the line width to `very thin`. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

2.8 Adding a Touch of Style

Instead of the options `gray,very thin` Karl could also have said `help lines`. *Styles* are predefined sets of options that can be used to organize how a graphic is drawn. By saying `help lines` you say "use the style that I (or someone else) has set for drawing help lines". If Karl decides, at some later point, that grids should be drawn, say, using the color `blue!50` instead of `gray`, he could provide the following option somewhere:

```
help lines/.style={color=blue!50,very thin}
```

The effect of this "style setter" is that in the current scope or environment the `help lines` option has the same effect as `color=blue!50,very thin`.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner. Normally, styles are defined at the beginning of a picture. However, you may sometimes wish to define a style globally, so that all pictures of your document can use this style. Then you can easily change the way all graphics look by changing this one style. In this situation you can use the `\tikzset` command at the beginning of the document as in

```
\tikzset{help lines/.style=very thin}
```

To build a hierarchy of styles you can have one style use another. So in order to define a style `Karl's grid` that is based on the `grid` style Karl could say

```
\tikzset{Karl's grid/.style={help lines,color=blue!50}}
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Styles are made even more powerful by parametrization. This means that, like other options, styles can also be used with a parameter. For instance, Karl could parameterize his grid so that, by default, it is blue, but he could also use another color.

```
\begin{tikzpicture}
  [Karl's grid/.style  ={help lines,color=#1!50},
   Karl's grid/.default=blue]

  \draw[Karl's grid]     (0,0) grid (1.5,2);
  \draw[Karl's grid=red] (2,0) grid (3.5,2);
\end{tikzpicture}
```

In this example, the definition of the style `Karl's grid` is given as an optional argument to the `{tikzpicture}` environment. Additional styles for other elements would follow after a comma. With many styles in effect, the optional argument of the environment may easily happen to be longer than the actual contents.

2.9 Drawing Options

Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

He saw that the style `very thin` yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that `thin` is the correct choice, since it gives the same thickness as TEX's `\hrule` command. Nevertheless, Karl would like to know whether there is anything "in the middle" between `thin` and `thick`. There is: `semithick`.

Another useful thing one can do with lines is to dash or dot them. For this, the two styles `dashed` and `dotted` can be used, yielding - - - - and ····. Both options also exist in a loose and a dense version, called `loosely dashed`, `densely dashed`, `loosely dotted`, and `densely dotted`. If he really, really needs to, Karl can also define much more complex dashing patterns with the `dash pattern` option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

2.10 Arc Path Construction

Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction operation is useful, which draws part of a circle or ellipse. This `arc` operation is followed by options in brackets that specify the arc. An example would be `arc[start angle=10, end angle=80, radius=10pt]`, which means exactly what it says. Karl obviously needs an arc from 0° to 30°. The radius should be something relatively small, perhaps around one third of the circle's radius. When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to "get there".