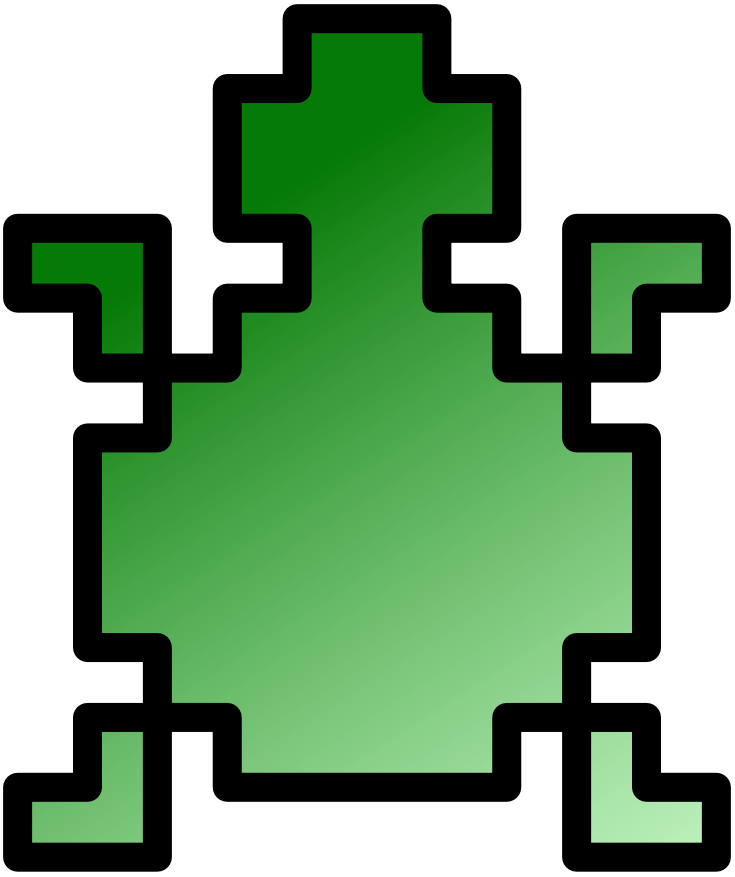
The KTurtle Handbook

Cies Breijs
Anne-Marie Mahfouf
Mauricio Piacentini



Contents

List of Tables

Abstract

KTurtle is an educational programming environment that aims to make learning how to program as easy as possible. To achieve this KTurtle makes all programming tools available from the user interface. The programming language used is TurtleScript which allows its commands to be translated.

Chapter 1

Introduction

KTurtle is an educational programming environment that uses TurtleScript, a programming language loosely based on and inspired by Logo. The goal of KTurtle is to make programming as easy and accessible as possible. This makes KTurtle suitable for teaching kids the basics of math, geometry and... programming. One of the main features of TurtleScript is the ability to translate the commands into the speaking language of the programmer.

KTurtle is named after 'the turtle' that plays a central role in the programming environment. The student will usually instruct the turtle, using the TurtleScript commands, to make a drawing on the canvas.

1.1 What is TurtleScript?

TurtleScript, the programming language used in KTurtle, is inspired by the Logo family of programming languages. The first version of Logo was created by Seymour Papert of MIT Artificial Intelligence Laboratory in 1967 as an offshoot of the LISP programming language. From then many versions of Logo have been released. By 1980 Logo was gaining momentum, with versions for MSX, Commodore, Atari, Apple II and IBM PC systems. These versions were mainly for educational purposes. The MIT is still maintains a website on Logo containing a list of several popular implementation of the language.

TurtleScript shares a feature found in many other implementations of Logo: the ability to translate the commands to suit the native language of the student. This feature makes it easier for students that have no or little understanding of English to get started. Besides this feature KTurtle has many other features aimed at easing the students initial experience with programming.

1.2 Features of KTurtle

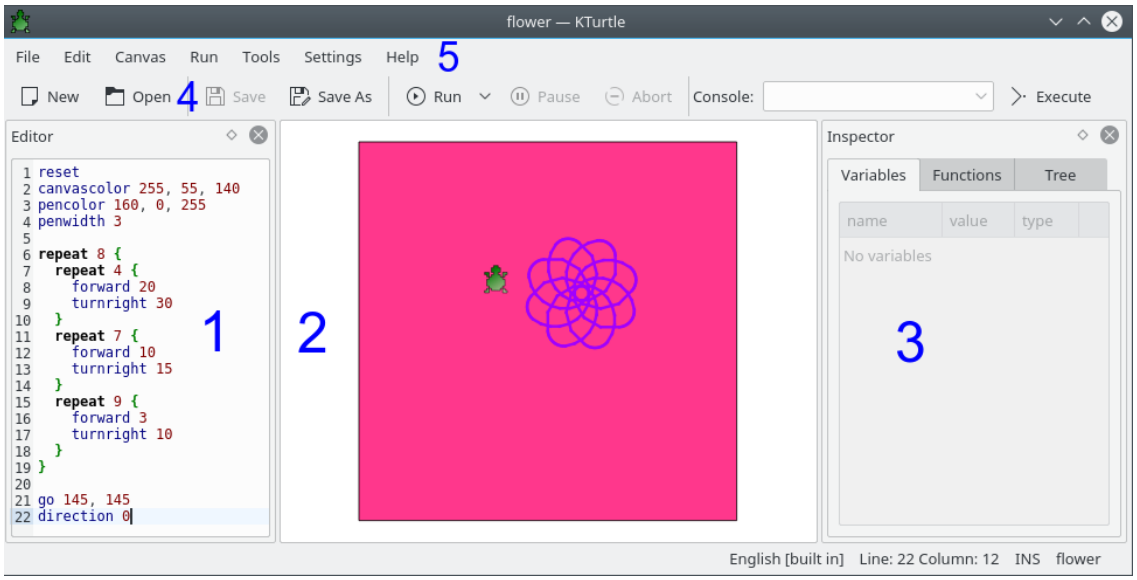
KTurtle has some nice features that make starting to program a breeze. See here some of the highlights of KTurtle feature set:

- An integrated environment with TurtleScript interpreter, editor, canvas and other tools all in one application (no extra dependencies).
- The ability to translate the TurtleScript commands using the translation framework of KDE.
- TurtleScript supports user defined functions, recursion and dynamic type switching.
- The execution can be slowed down, paused or stopped at any time.

- A powerful editor featuring intuitive syntax highlighting, line numbering, error markers, visual execution and more.
- The canvas, where the turtle draws, can be printed or saved either as an image (PNG) or a drawing (SVG).
- Context help: help where you need it. Just press **F2** (or see **Help** → **Help on: ...**) to get help on the piece of code currently under your cursor.
- An error dialog that links the error messages to the mistakes in the program and marks them red.
- Simplified programming terminology.
- Integrated example programs to make it easy to get started. These examples are translated using KDE translation framework.

Chapter 2

Using KTurtle



The main window of KTurtle has three main parts: the editor (1) on the left where you type the TurtleScript commands, the the canvas (2) on the right where the turtle make your drawing, and the inspector (3) which gives you information when your program executes. Besides these you find the menu bar (5) from where all the actions can be reached, the toolbar (4) that allows you to quickly select the most used actions, the **Console**, that you can use to enter a one line command to test it, and the statusbar (along the bottom of the window) where you will find feedback on the state of KTurtle.

2.1 The Editor

In the editor you type the TurtleScript commands. Most of functions of the editor can be found in the File and Edit menus. The editor can be docked on each border of the main window or it can be detached and placed anywhere on your desktop.

You have several ways to get some code in the editor. The easiest way is to use an example: choose **File** → **Examples** in the File menu and select an example. The file example you choose will be opened in the the editor, you can then use **Run** → **Run** from the menubar or the **Run** from the toolbar to run the code if you like.

You can open TurtleScript files by choosing **File** → **Open...**.

The third way is to directly type your own code in the editor or to copy/paste some code.

2.2 The Canvas

The canvas is the domain of the turtle, here the turtle draws according to the commands it gets. After getting some code in the Editor and executing it, two things can happen: either the code executes fine, and will you most likely see something change on the canvas; or you have made an error in your code in that case the error tab will appear explaining you what mistake you made.

You can zoom in and out the canvas with your mouse wheel.

2.3 The Inspector

The inspector informs you about the variables, the learned functions and show the code tree while the program is running.

The inspector can be docked on each border of the main window or it can be detached and placed anywhere on your desktop.

2.4 The Toolbar

Here you can quickly reach the most used actions. The Toolbar also contains the **Console** where you can quickly invoke commands, this might be useful in case you want to test a command without modifying the content of the Editor.

You can configure the toolbar using **Settings** → **Configure Toolbars...** to better fit your preferences.

2.5 The Menubar

In the menubar you find all the actions of KTurtle. They are in the following groups: **File**, **Edit**, **Canvas**, **Run**, **Tools**, **Settings**, and **Help**. This section describes them all.

2.5.1 The File Menu

File → **New (Ctrl-N)**

Creates a new, empty TurtleScript file.

File → **Open... (Ctrl-O)**

Opens a TurtleScript file.

File → **Open Recent**

Opens a TurtleScript file that has been opened recently.

File → Examples

Open example TurtleScript programs. The examples are in your favorite language that you can choose in **Settings → Script Language**.

File → Get more examples...

Open the get **Hot New Stuff** dialog to download additional TurtleScript files from the Internet.

File → Save (Ctrl-S)

Saves the currently opened TurtleScript file.

File → Save As... (Ctrl-Shift-S)

Saves the currently opened TurtleScript file on a specified location.

File → Export to HTML...

Exports the current content of the Editor as an HTML file that includes highlighting colors.

File → Print... (Ctrl-P)

Prints the current code in the editor.

File → Quit (Ctrl-Q)

Quits KTurtle.

2.5.2 The Edit Menu

Edit → Undo (Ctrl-Z)

Undoes the last change to code. KTurtle has unlimited undos.

Edit → Redo (Ctrl-Shift-Z)

Redoes an undone change to the code.

Edit → Cut (Ctrl-X)

Cuts the selected text from the editor to the clipboard.

Edit → Copy (Ctrl-C)

Copies the selected text from the editor to the clipboard.

Edit → Paste (Ctrl-V)
Pastes the text from the clipboard to the editor.

Edit → Select All (Ctrl-A)
Selects all the text from the editor.

Edit → Find... (Ctrl-F)
With this action you can find phrases in the code.

Edit → Find Next (F3)
Use this to find the next occurrence of the phrase you searched for.

Edit → Find Previous (Shift-F3)
Use this to find the previous occurrence of the phrase you searched for.

Edit → Overwrite Mode (Ins)
Toggle between the 'insert' and 'overwrite' mode.

2.5.3 The Canvas Menu

Canvas → Export to Image (PNG)...
Exports the current content of the Canvas as a raster image of the PNG (Portable Network Graphics) type.

Canvas → Export to Drawing (SVG)...
Exports the current content of the Canvas as a vector drawing of the SVG (Scalable Vector Graphics) type.

Canvas → Print Canvas...
Prints the current content of the Canvas.

2.5.4 The Run Menu

Run → Run (F5)
Starts the execution of the commands in the editor.

Run → Pause (F6)
Pauses the execution. This action is only enabled when the commands are actually executing.

Run → Abort (F7)

Stops the execution. This action is only enabled when the commands are actually executing.

Run → Run Speed

Present a list of possible execution speeds, consisting of: **Full Speed (no highlighting and inspector)**, **Full Speed**, **Slow**, **Slower**, **Slowest** and **Step-by-Step**. When the execution speed is set to **Full Speed** (default) we can barely keep up with what is happening. Sometimes this behavior is wanted, but sometimes we want to keep track of the execution. In the latter case you want to set the execution speed to **Slow**, **Slower** or **Slowest**. When one of the slow modes is selected the current position of the executor will be shown in the editor. **Step-by-Step** will execute one command at a time.

2.5.5 The Tools Menu

Tools → Direction Chooser...

This action opens the direction chooser dialog.

Tools → Color Picker...

This action opens the color picker dialog.

2.5.6 The Settings Menu

Settings → Script Language

Choose the language for the code.

Settings → Show Editor (Ctrl-E)

Show or hide the Editor.

Settings → Show Inspector (Ctrl-I)

Show or hide the inspector.

Settings → Show Errors

Show or hide the **Error** tab with a list of errors resulting from running the code. If this option is enabled, click on **Canvas** to see the turtle again.

Settings → Show Line Numbers (F11)

With this action you can show the line numbers in the editor. This can be handy for finding errors.

Settings → **Show Toolbar**

Toggle the Main Toolbar

Settings → **Show Statusbar**

Toggle the Statusbar

Settings → **Configure Shortcuts...**

Standard KDE dialog to configure the shortcuts.

Settings → **Configure Toolbars...**

The standard KDE dialog for configuring the toolbars.

2.5.7 The Help Menu

KTurtle has a default KDE **Help** menu as described in the KDE Fundamentals with one additional entry:

Help → **Help on: ... (F2)**

This is a very useful function: it provides help on the code where the cursor in the editor is at. So, e.g., you have used the **`print`** command in your code, and you want to read and to know what the handbook says on this command. You just move your cursor so it is in the **`print`** command and you press **F2**. The handbook will then show all info on the **`print`** command.

This function can prove to be useful while learning TurtleScript.

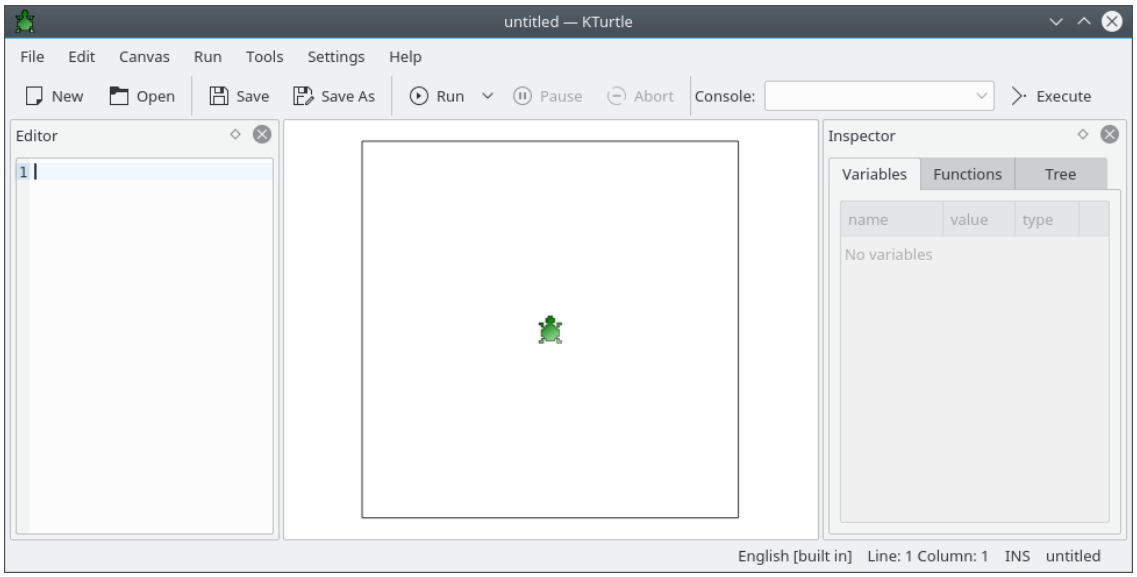
2.6 The Statusbar

On the status bar you get feedback of the state of KTurtle. On the left side it shows the feedback on the last action. On the right side you find the current location of the cursor (line and column numbers). In the middle of the status bar is indicated the current language used for the commands.

Chapter 3

Getting Started

When you start KTurtle you will see something like this:



In this Getting Started guide we assume that the language of the TurtleScript commands is English. You can change this language with **Settings** → **Script Language**. Be aware that the language you set here for KTurtle is the one you use to type the TurtleScript commands, not the language used by KDE on your computer and used to display the KTurtle interface and menus.

3.1 First steps with TurtleScript: meet the Turtle!

You must have noticed the turtle in the middle of the canvas: you are just about to learn how to control it using commands in the editor.

3.1.1 The Turtle Moves

Let us start by getting the turtle moving. Our turtle can do 3 types of moves, (1) it can move forwards and backwards, (2) it can turn left and right and (3) it can go (jump) directly to a position on the screen. Try this for example:

```
forward 100
turnleft 90
```

Type or copy-paste the code to the editor and execute it (using **Run** → **Run**) to see the result.

When you typed and executed the commands like above in the editor you might have noticed one or more of the following things:

1. That — after executing the commands — the turtle moves up, draws a line, and then turns a quarter turn to the left. This because you have used the **forward** and the **turnleft** commands.
2. That the color of the code changed while you where typing it: this feature is called *intuitive highlighting* — different types of commands are highlighted differently. This makes reading large blocks of code more easy.
3. That the turtle draws a thin black line.
4. Maybe you got an error message. This could simply mean two things: you could have made a mistake while copying the commands, or you should still set the correct language for the TurtleScript commands (you can do that by choosing **Settings** → **Script Language**).

You will likely understand that **forward 100** instructed the turtle to move forward leaving a line, and that **turnleft 90** instructed the turtle to turn 90 degrees to the left.

Please see the following links to the reference manual for a complete explanation of the new commands: **forward**, **backward**, **turnleft**, and **turnright**.

3.1.2 More examples

The first example was very simple, so let us go on!

```
reset

canvassize 200,200
canvascolor 0,0,0
pencolor 255,0,0
penwidth 5

go 20,20
direction 135

forward 200
turnleft 135
forward 100
turnleft 135
forward 141
turnleft 135
forward 100
turnleft 45

go 40,100
```

Again you can type or copy-paste the code to the editor or open the `arrow` example in the **Examples** menu and execute it (using **Run** → **Run**) to see the result. In the next examples you are expected to know the drill.

You might have noticed that this second example uses a lot more code. You have also seen a couple of new commands. Here a short explanation of all the new commands:

After a **reset** command everything is like is was when you had just started KTurtle.

canvassize 200,200 sets the canvas width and height to 200 pixels. The width and the height are equal, so the canvas will be a square.

canvascolor 0,0,0 makes the canvas black. **0,0,0** is a RGB-combination where all values are set to **0**, which results in black.

pencolor 255,0,0 sets the color of the pen to red. **255,0,0** is a RGB-combination where only the red value is set to **255** (fully on) while the others (green and blue) are set to **0** (fully off). This results in a bright shade of red.

If you do not understand the color values, be sure to read the glossary on RGB-combination.

penwidth 5 sets the width (the size) of the pen to **5** pixels. From now on every line the turtle draw will have a thickness of **5**, until we change the **penwidth** to something else.

go 20,20 commands the turtle to go to a certain place on the canvas. Counted from the upper left corner, this place is 20 pixels across from the left, and 20 pixels down from the top of the canvas. Note that using the **go** command the turtle will not draw a line.

direction 135 set the turtle's direction. The **turnleft** and **turnright** commands change the turtle's angle starting from its current direction. The **direction** command changes the turtle's angle from zero, and thus is not relative to the turtle previous direction.

After the **direction** command a lot of **forward** and **turnleft** commands follow. These command do the actual drawing.

At last another **go** command is used to move the turtle aside.

Make sure you follow the links to the reference. The reference explains each command more thoroughly.

Chapter 4

TurtleScript Programming Reference

This is the reference for KTurtle's TurtleScript. In the first section of this chapter have a look at some aspects of the grammar of TurtleScript programs. The second section deals exclusively with mathematical operators, boolean (true/false) operators and comparison operators. The third section is basically a giant list of all commands explaining them one-by-one. Section four explains how to assign values to variables. Finally we explain how to arrange the execution of commands with execution controlling statements in section five and how to create you own commands with learn in section six.

4.1 The Grammar of TurtleScript

As in any language, TurtleScript has different types of words and symbols. In English we distinguish verbs (like 'to walk' or 'to sing') and nouns (like 'sister' or 'house'), they are used for different purposes. TurtleScript is a programming language, it is used to instruct KTurtle what to do.

In this section some of TurtleScript's different types of words and symbols are briefly explained. We explain comments, commands and the three different kinds of literals: numbers, strings and boolean (true/false) values.

4.1.1 Comments

A program consists instructions that are executed when the program is run and so called comments. Comments are not executed, KTurtle simply ignores them when executing your program. Comment are there for other programmers to make them understand your program better. Everything that follows on a **#** symbol is considered a comment in TurtleScript. For example this little program that does nothing:

```
# this little program does nothing, it is only a comment!
```

It is a bit useless but it explain the matter well.

Comments get very useful when the program gets a little bit more complex. It can help to give some advice to other programmers. In the following program you see comments being used together with the print command.

```
# this program has been made by Cies Breijs.
print "this text will get printed on the canvas"
# the previous line is not a comment, but the next line is:
# print "this text will not get printed!"
```

The first line describes the program. The second line is executed by KTurtle and prints **`this text will get printed on the canvas`** on the canvas. The third line is a comment. And the forth line is a comment that contains a piece of TurtleScript, if the **`#`** symbol would be removed on the fourth line the print statement will we executed by KTurtle. Programmers say: the print statement on the fourth line is 'commented out'.

Commented lines are highlighted with light gray in the code editor.

4.1.2 Commands

Using commands you tell the turtle or KTurtle to do something. Some commands need input, some give output.

```
# forward is a command that needs input, in this case the number 100:
forward 100
```

The first line is a comment. The second line contains the **`forward`** command and the number **`100`**. The number is not part of command, it is considered 'input' for the command.

Some commands like e.g. **`go`** need more than one input value. Multiple values have to be separated using the **`,`** character (comma).

For a detailed overview of all commands that KTurtle supports go here. Built-in commands are highlighted in dark blue

4.1.3 Numbers

Most likely you already know quite a bit about numbers. The way numbers are used in KTurtle is not much different from spoken language, or math.

We have the so called natural numbers: **`0`**, **`1`**, **`2`**, **`3`**, **`4`**, **`5`**, etc. The negative numbers: **`-1`**, **`-2`**, **`-3`**, etc. And the numbers with decimals, or dot-numbers, for example: **`0.1`**, **`3.14`**, **`33.3333`**, **`-5.05`**, **`-1.0`**. The **`.`** character (dot) is used as decimal separator.

Numbers can be used in mathematical operators and comparison operators. They can also be stored in variables. Numbers are highlighted in dark red.

4.1.4 Strings

First an example:

```
print "Hello, I'm a string."
```

In this example **`print`** is a command where **`´´Hello, I'm a string.´´`** is a string. Strings start and end with the **`´´`** mark, by these marks KTurtle knows it is a string.

Strings can be put in variables, just like numbers. Yet, unlike numbers, strings cannot be used in mathematical operators or comparison operators. Strings are highlighted with red.

4.1.5 Boolean (true/false) values

There are only two boolean values: **`true`** and **`false`**. Sometimes they are also called: on and off, yes and no, one and zero. But in TurtleScript we call them, always, **`true`** and **`false`**. Have a look at this piece of TurtleScript:

```
$a = true
```

If you look in the inspector you can see that the variable **$a** is set to **true**, and has the boolean type.

Often boolean values are the result of a comparison operator, like in the following piece of TurtleScript:

```
$answer = 10 > 3
```

The variable **$answer** is set to **true** because **10** is larger than **3**.

Boolean values, **true** and **false**, are highlighted with dark red.

4.2 Mathematical, boolean and comparing operators

The title of this section might sound very difficult, yet it is not as difficult as it sound.

4.2.1 Mathematical operators

These are the basic math symbols known as: add (**+**), subtract (**-**), multiply (*****), divide (**/**) and power (^).

Here a small example of the mathematical operators you can use in TurtleScript:

```
$add      = 1 + 1
$subtract = 20 - 5
$multiply = 15 * 2
$divide   = 30 / 30
$power    = 2 ^ 2
```

The values resulting from the mathematical operations get assigned to various variables. Using the inspector you can see the values.

If you just want a simple calculation to be done you can do something like this:

```
print 2010-12
```

Now an example with parentheses:

```
print ( ( 20 - 5 ) * 2 / 30 ) + 1
```

The expressions inside parentheses will be calculated first. In this example, 20-5 will be calculated, then multiplied by 2, divided by 30, and then 1 is added (giving 2). Parentheses can also be used in other cases.

KTurtle also has more advanced mathematical features in the form of commands. Have a look at the following commands but be aware that it concerns advanced operations: round, random, sqrt , pi, sin, cos, tan, arcsin, arccos, arctan.

4.2.2 Boolean (true/false) operators

Where mathematical operators are mainly for numbers, boolean operators are for boolean values (**true** and **false**). There are only three boolean operators, namely: **and**, **or**, and **not**. The following piece of TurtleScript shows how to use them:

```
$and_1_1 = true and true    # -> true
$and_1_0 = true and false   # -> false
$and_0_1 = false and true   # -> false
$and_0_0 = false and false  # -> false

$or_1_1 = true or true    # -> true
$or_1_0 = true or false   # -> true
$or_0_1 = false or true   # -> true
$or_0_0 = false or false  # -> false

$not_1 = not true   # -> false
$not_0 = not false  # -> true
```

Using the inspector you can see the values, yet we also supply these results as little comments at the end of the lines. **and** evaluates **true** only if both sides are **true**. **or** evaluates **true** if either side is **true**. And **not** turns a **true** into **false** and a **false** into **true**.

Boolean operators are highlighted with pink.

4.2.2.1 Some more advanced examples

Consider the following example with **and**:

```
$a = 1
$b = 5
if (($a < 10) and ($b == 5)) and ($a < $b) {
  print "hello"
}
```

In this piece of TurtleScript the result of three comparing operators are merged using **and** operators. This means that all three have to evaluate ˝true˝ in order for the ˝hello˝ to be printed.

An example with **or**:

```
$n = 1
if ($n < 10) or ($n == 2) {
  print "hello"
}
```

In this piece of TurtleScript the left side of the **or** is evaluating to 'true', the right side to 'false'. Since one of the two sides of the **or** operator is 'true', the **or** operator evaluates 'true'. That means ˝hello˝ gets printed.

And finally an example with **not** which changes 'true' into 'false' and 'false' into 'true'. Have a look:

```
$n = 1
if not ($n == 3) {
  print "hello"
} else {
  print "not hello ;-)"
}
```

4.2.3 Comparing operators

Consider this simple comparison:

```
$answer = 10 > 3
```

Here **10** is compared to **3** with the 'greater than' operator. The result of this comparison, the boolean value **true** is stored in the variable **$answer**.

All numbers and variables (that contain numbers) can be compared to each other with comparing operators.

Here are all possible comparing operators:

$A == $B	equals	answer is 'true' if **$A** equals **$B**
$A != $B	not-equals	answer is 'true' if **$A** does not equal **$B**
$A > **$B**	greater than	answer is 'true' if **$A** is greater than **$B**
$A < **$B**	smaller than	answer is 'true' if **$A** is smaller than **$B**
$A >= $B	greater than or equals	answer is 'true' if **$A** is greater than or equals **$B**
$A <= $B	smaller than or equals	answer is 'true' if **$A** is smaller than or equals **$B**

Table 4.1: Types of questions

Please note that $A and $B have to be numbers or variables that contain numbers.

4.3 Commands

Using commands you tell the turtle or KTurtle to do something. Some commands need input, some give output. In this section we explain all the built-in commands of KTurtle. Alternatively, using learn, you can create your own commands. Built-in commands we discuss here are highlighted with dark blue.

4.3.1 Moving the turtle

There are several commands to move the turtle over the screen.

forward (fw)

```
forward X
```

forward moves the turtle forward by the amount of X pixels. When the pen is down the turtle will leave a trail. **forward** can be abbreviated to **fw**

backward (bw)

```
backward X
```

backward moves the turtle backward by the amount of X pixels. When the pen is down the turtle will leave a trail. **backward** can be abbreviated to **bw**.

turnleft (tl)

```
turnleft X
```

turnleft commands the turtle to turn an amount of X degrees to the left. **turnleft** can be abbreviated to **tl**.

turnright (tr)

```
turnright X
```

turnright the turtle to turn an amount of X degrees to the right. **turnright** can be abbreviated to **tr**.

direction (dir)

```
direction X
```

direction set the turtle's direction to an amount of X degrees counting from zero, and thus is not relative to the turtle's previous direction. **direction** can be abbreviated to **dir**.

getdirection

```
getdirection
```

getdirection returns the turtle's direction as an amount of degrees counting from zero, where zero is the direction when the turtle is pointing upwards.

center

```
center
```

center moves the turtle to the center on the canvas.

go

```
go X,Y
```

go commands the turtle to go to a certain place on the canvas. This place is X pixels from the left of the canvas, and Y pixels from the top of the canvas.

gox

```
gox X
```

gox using this command the turtle will move to X pixels from the left of the canvas whilst staying at the same height. **gox** can be abbreviated to **gx**.

goy

```
goy Y
```

goy using this command the turtle will move to Y pixels from the top of the canvas whilst staying at the same distance from the left border of the canvas. **goy** can be abbreviated to **gy**.

> NOTE
> Using the commands **go**, **gox**, **goy** and **center** the turtle will not draw a line, no matter if the pen is up or down.

4.3.2 Where is the turtle?

There are two commands which return the position of the turtle on the screen.

getx

getx returns the number of pixels from the left of the canvas to the current position of the turtle.

gety

gety returns the number of pixels from the top of the canvas to the current position of the turtle.

4.3.3 The turtle has a pen

The turtle has a pen that draws a line when the turtle moves. There are a few commands to control the pen. In this section we explain these commands.

penup (pu)

```
penup
```

penup lifts the pen from the canvas. When the pen is 'up' no line will be drawn when the turtle moves. See also **pendown**. **penup** can be abbreviated to **pu**.

pendown (pd)

```
pendown
```

pendown presses the pen down on the canvas. When the pen is press 'down' on the canvas a line will be drawn when the turtle moves. See also **penup**. **pendown** can be abbreviated to **pd**.

penwidth (pw)

```
penwidth X
```

penwidth sets the width of the pen (the line width) to an amount of X pixels. **penwidth** can be abbreviated to **pw**.

pencolor (pc)

```
pencolor R,G,B
```

pencolor sets the color of the pen. **pencolor** takes an RGB combination as input. **pencolor** can be abbreviated to **pc**.

4.3.4 Commands to control the canvas

There are several commands to control the canvas.

canvassize (cs)

```
canvassize X,Y
```

With the **canvassize** command you can set the size of the canvas. It takes X and Y as input, where X is the new canvas width in pixels, and Y is the new height of the canvas in pixels. **canvassize** can be abbreviated to **cs**.

canvascolor (cc)

```
canvascolor R,G,B
```

canvascolor set the color of the canvas. **canvascolor** takes an RGB combination as input. **canvascolor** can be abbreviated to **cc**.

4.3.5 Commands to clean up

There are two commands to clean up the canvas after you have made a mess.

clear (ccl)

```
clear
```

With **clear** you can clean all drawings from the canvas. All other things remain: the position and angle of the turtle, the canvascolor, the visibility of the turtle, and the canvas size.

reset

```
reset
```

reset cleans much more thoroughly than the **clear** command. After a **reset** command everything is like is was when you had just started KTurtle. The turtle is positioned at the middle of the screen, the canvas color is white, the turtle draws a black line on the canvas and the canvassize is set to 400 x 400 pixels.

4.3.6 The turtle is a sprite

First a brief explanation of what sprites are: sprites are small pictures that can be moved around the screen, like we often see in computer games. Our turtle is also a sprite. For more info see the glossary on sprites.

Next you will find a full overview on all commands to work with sprites.

[The current version of KTurtle does not yet support the use of sprites other than the turtle. With future versions you will be able to change the turtle into something of your own design]

spriteshow (ss)

```
spriteshow
```

spriteshow makes the turtle visible again after it has been hidden. **spriteshow** can be abbreviated to **ss**.

spritehide (sh)

```
spritehide
```

spritehide hides the turtle. This can be used if the turtle does not fit in your drawing. **spritehide** can be abbreviated to **sh**.

4.3.7 Can the turtle write?

The answer is: 'yes'. The turtle can write: it writes just about everything you command it to.

print

```
print X
```

The **print** command is used to command the turtle to write something on the canvas. **print** takes numbers and strings as input. You can **print** various numbers and strings using the '+' symbol. See here a small example:

```
$year = 2003
$author = "Cies"
print $author + " started the KTurtle project in " + $year + " and  ↩
    still enjoys working on it!"
```

fontsize

```
fontsize X
```

fontsize sets the size of the font that is used by **print**. **fontsize** takes one input which should be a number. The size is set in pixels.

4.3.8 Mathematical commands

The following commands are KTurtle's more advanced mathematical commands.

round

```
round(x)
```

round the given number to the nearest integer.

```
print round(10.8)
forward 20
print round(10.3)
```

With this code the turtle will print the numbers 11 and 10.

random (rnd)

```
random X,Y
```

random is a command that takes input and gives output. As input are required two numbers, the first (X) sets the minimum output, the second (Y) sets the maximum. The output is a randomly chosen number that is equal or greater than the minimum and equal or smaller than the maximum. Here a small example:

```
repeat 500 {
  $x = random 1,20
  forward $x
  turnleft 10 - $x
}
```

Using the **random** command you can add a bit of chaos to your program.

mod

```
mod X,Y
```

The **mod** returns remainder of the division of first number by the second number.

sqrt

```
sqrt X
```

The **sqrt** command is sued to find the square root of a number, X.

pi

```
pi
```

This command returns the constant Pi, **3.14159**.

sin, cos, tan

```
sin X
cos X
tan X
```

These three commands represent the world famous trigoniometrical functions **sin**, **cos** and **tan**. The input argument of these commands, X, is a number.

arcsin, arccos, arctan

```
arcsin X
arccos X
arctan X
```

These commands are the inverse functions of sin, cos and tan. The input argument of these commands, X, is a number.

4.3.9 Input and feedback through dialogs

A dialog is a small pop-up window that provides some feedback or asks for some input. KTurtle has two commands for dialogs, namely: **message** and **ask**

message

```
message X
```

The **message** command takes a string as input. It shows a pop-up dialog containing the text from the string.

```
message "Cies started KTurtle in 2003 and still enjoys working on it!"
```

ask

```
ask X
```

ask takes a string as input. It shows this string in a pop-up dialog (similar to message), along with an input field. After the user has entered a number or a string into this, the result can be stored in a variable or passed as an argument to a command. For example:

```
$in = ask "What is your year of birth?"
$out = 2003 - $in
print "In 2003 you were " + $out + " years old at some point."
```

If the user cancels the input dialog, or does not enter anything at all, the variable is empty.

4.4 Assignment of variables

First we have a look at variables, then we look at assigning values to those variables.

Variables are words that start with a '$', in the editor they are highlighted with purple.

Variables can contain any number, string or boolean (true/false) value. Using the assignment, **=**, a variable is given its content. It will keep that content until the program finishes executing or until the variable is reassigned to something else.

You can use variables, once assigned, just as if they are their content. For instance in the following piece of TurtleScript:

```
$x = 10
$x = $x / 3
print $x
```

First the variable **$x** is assigned to **10**. Then **$x** is reassigned to itself divided by **3** — this effectively means **$x** is reassigned to product of **10 / 3**. Finally **$x** is printed. In line two and three you see that **$x** is used as if it is its contents.

Variables have to be assigned in order to be used. For example:

```
print $n
```

Will result in an error message.

Please consider the following piece of TurtleScript:

```
$a = 2004
$b = 25

# the next command prints "2029"
print $a + $b
backward 30
# the next command prints "2004 plus 25 equals 2029"
print $a + " plus " + $b + " equals " + ($a + $b)
```

In the first two lines the variables **$a** and **$b** are set to 2004 and 25. Then in two **print** commands with a **backward 30** in between are executed. The comments before the **print** commands explain what they are doing. The command **backward 30** is there to make sure every output is on a new line. As you see variables can be used just as if their where what they contain, you can use them with any kind of operators or give them as input when invoking commands.

One more example:

```
$name = ask "What is your name?"
print "Hi " + $name + "! Good luck while learning the art of programming ↩
    ..."
```

Pretty straight forward. Again you can see that the variable **$name**, treated just like a string.

When using variables the inspector is very helpful. It shows you the contents of all variables that are currently in use.

4.5 Controlling execution

The execution controllers enable you — as their name implies — to control execution.

Execution controlling commands are highlighted with dark green in a bold font type. The brackets are mostly used together with execution controllers and they are highlighted with black.

4.5.1 Have the turtle wait

If you have done some programming in KTurtle you have might noticed that the turtle can be very quick at drawing. This command makes the turtle wait for a given amount of time.

wait

```
wait X
```

wait makes the turtle wait for X seconds.

```
repeat 36 {
  forward 5
  turnright 10
  wait 0.5
}
```

This code draws a circle, but the turtle will wait half a second after each step. This gives the impression of a slow-moving turtle.

4.5.2 Execute ˝if˝

if

```
if boolean { ... }
```

The code that is placed between the brackets will only be executed **if** the boolean value evaluates 'true'.

```
$x = 6
if $x > 5 {
  print "$x is greater than five!"
}
```

On the first line **$x** is set to 6. On the second line a comparing operator is used to evaluate **$x** > **5**. Since this evaluates 'true', 6 is larger than 5, the execution controller **if** will allow the code between the brackets to be executed.

4.5.3 If not, in other words: ˝else˝

else

```
if boolean { ... } else { ... }
```

else can be used in addition to the execution controller **if**. The code between the brackets after **else** is only executed if the boolean evaluates 'false'.

```
reset
$x = 4
if $x > 5 {
  print "$x is greater than five!"
} else {
  print "$x is smaller than six!"
}
```

The comparing operator evaluates the expression **$x** > **5**. Since 4 is not greater than 5 the expression evaluates 'false'. This means the code between the brackets after **else** gets executed.

4.5.4 The ˝while˝ loop

while

```
while boolean { ... }
```

The execution controller **while** is a lot like **if**. The difference is that **while** keeps repeating (looping) the code between the brackets until the boolean evaluates 'false'.

```
$x = 1
while $x < 5 {
  forward 10
  wait 1
  $x = $x + 1
}
```

On the first line **$x** is set to 1. On the second line **$x** < **5** is evaluated. Since the answer to this question is 'true' the execution controller **while** starts executing the code between the brackets until the **$x** < **5** evaluates 'false'. In this case the code between the brackets will be executed 4 times, because every time the fifth line is executed **$x** increases by 1.

4.5.5 The ˝repeat˝ loop

repeat

```
repeat number { ... }
```

The execution controller **repeat** is a lot like **while**. The difference is that **repeat** keeps repeating (looping) the code between the brackets for as many times as the given number.

4.5.6 The ˝for˝ loop, a counting loop

for

```
for variable = number to number { ... }
```

The **for** loop is a 'counting loop', i.e. it keeps count for you. The first number sets the variable to the value in the first loop. Every loop the number is increased until the second number is reached.

```
for $x = 1 to 10 {
  print $x * 7
  forward 15
}
```

Every time the code between the brackets is executed the **$x** is increased by 1, until **$x** reaches the value of 10. The code between the brackets prints the **$x** multiplied by 7. After this program finishes its execution you will see the times table of 7 on the canvas.

The default step size of a loop is 1, you can use an other value with

```
for variable = number to number step number { ... }
```

4.5.7 Leave a loop

break

```
break
```

Terminates the current loop immediately and transfers control to the statement immediately following that loop.

4.5.8 Stop executing your program

exit

```
exit
```

Finishes the execution of your program.

4.5.9 Checking assertions at runtime

assert

```
assert boolean
```

Can be used to reason about program or input correctness.

```
$in = ask "What is your year of birth?"
# the year must be positive
assert $in > 0
```

4.6 Create your own commands with 'learn'

learn is special as it is used to create your own commands. The commands you create can take input and return output. Let us take a look at how a new command is created:

```
learn circle $x {
  repeat 36 {
    forward $x
    turnleft 10
  }
}
```

The new command is called **circle**. **circle** takes one input argument, to set the size of the circle. **circle** returns no output. The **circle** command can now be used like a normal command in the rest of the code. See this example:

```
learn circle $X {
  repeat 36 {
    forward $X
    turnleft 10
  }
}

go 200,200
circle 20

go 300,200
circle 40
```

In the next example, a command with a return value is created.

```
learn faculty $x {
  $r = 1
  for $i = 1 to $x {
    $r = $r * $i
  }
  return $r
}

print faculty 5
```

In this example a new command called **`faculty`** is created. If the input of this command is **`5`** then the output is **`5*4*3*2*1`**. By using **`return`** the output value is specified and the execution is returned.

Commands can have more than one input. In the next example, a command that draws a rectangle is created:

```
learn box $x, $y {
  forward $y
  turnright 90
  forward $x
  turnright 90
  forward $y
  turnright 90
  forward $x
  turnright 90
}
```

Now you can run **`box 50, 100`** and the turtle will draw a rectangle on the canvas.

Chapter 5

Glossary

In this chapter you will find an explanation of most of the 'uncommon' words that are used in the handbook.

degrees

Degrees are units to measure angles or turns. A full turn is 360 degrees, a half turn 180 degrees and a quarter turn 90 degrees. The commands **turnleft**, **turnright** and **direction** need an input in degrees.

input and output of commands

Some commands take input, some commands give output, some commands take input *and* give output and some commands neither take input nor give output.

Some examples of commands that only take input are:

```
forward 50
pencolor 255,0,0
print "Hello!"
```

The **forward** command takes **50** as input. **forward** needs this input to know how many pixels it should go forward. **pencolor** takes a color as input and **print** takes a string (a piece of text) as input. Please note that the input can also be a container. The next example illustrates this:

```
$x = 50
print $x
forward 50
$str = "hello!"
print $str
```

Now some examples of commands that give output:

```
$x = ask "Please type something and press OK... thanks!"
$r = random 1,100
```

The **ask** command takes a string as input, and outputs the number or string that is entered. As you can see, the output of **ask** is stored in the container **x**. The **random** command also gives output. In this case it outputs a number between 1 and 100. The output of the random is again stored in a container, named **r**. Note that the containers **x** and **r** are not used in the example code above.

There are also commands that neither need input nor give output. Here are some examples:

```
clear
penup
```

intuitive highlighting

This is a feature of KTurtle that makes coding even easier. With intuitive highlighting the code that you write gets a color that indicates what type of code it is. In the next list you will find the different types of code and the color they get in the editor.

regular commands	dark blue	The regular commands are described here.
execution controlling commands	black (bold)	These special commands control execution, read more on them here.
comments	gray	Lines that are commented start with a comment characters (#). These lines are ignored when the code is executed. Comments allow the programmer to explain a bit about his code or can be used to temporarily prevent a certain piece of code from executing.
brackets {, }	dark green (bold)	Brackets are used to group portions of code. Brackets are often used together with execution controllers.
the learn command	light green (bold)	The learn command is used to create new commands.
strings	red	Not much to say about (text) strings either, except that they always start and end with the double quotes (˝).
numbers	dark red	Numbers, well not much to say about them.
boolean values	dark red	There are exactly two boolean values, namely: true and false.
variables	purple	Start with a '$' and can contain numbers, strings or boolean values.
mathematical operators	gray	These are the mathematical operators: **+**, **−**, *****, **/** and **^**.
comparison operators	light blue (bold)	These are the comparison operators: **==**, **!=**, **<**, **>**, **<=** and **>=**.

boolean operators	pink (bold)	These are the boolean operators: **and**, **or** and **not**.
regular text	black	

Table 5.1: Different types of code and their highlight color

pixels

A pixel is a dot on the screen. If you look very close you will see that the screen of your monitor uses pixels. All images on the screen are built with these pixels. A pixel is the smallest thing that can be drawn on the screen.

A lot of commands need a number of pixels as input. These commands are: **forward**, **backward**, **go**, **gox**, **goy**, **canvassize** and **penwidth**.

In early versions of KTurtle the canvas was essentially a raster image, yet for recent versions the canvas is a vector drawing. This means that the canvas can be zoomed in and out, therefore a pixel does not necessarily have to translate to one dot on the screen.

RGB combinations (color codes)

RGB combinations are used to describe colors. The 'R' stand for 'red', the 'G' stands for 'green' and the 'B' stands for 'blue'. An example of an RGB combination is **255,0,0**: the first value ('red') is 255 and the others are 0, so this represents a bright shade of red. Each value of an RGB combination has to be in the range 0 to 255. Here a small list of some often used colors:

0,0,0	black
255,255,255	white
255,0,0	red
150,0,0	dark red
0,255,0	green
0,0,255	blue
0,255,255	light blue
255,0,255	pink
255,255,0	yellow

Table 5.2: Often used RGB combinations

Two commands need an RGB combination as input: these commands are **canvascolor** and **pencolor**.

sprite

A sprite is a small picture that can be moved around the screen. Our beloved turtle, for instance, is a sprite.

Note: with this version of KTurtle the sprite cannot be changed from a turtle into something else. Future versions of KTurtle will be able to do this.

Chapter 6

Translator's Guide to KTurtle

As you probably already know KTurtle's programming language, TurtleScript, allows to be translated. This takes away a barrier for some, especially younger students, on their effort to understand the basics of programming.

When translating KTurtle to a new language you will find, in addition to the GUI strings, the programming commands, the examples and the error messages are included in the standard .pot files as used for translation in KDE. Everything is translated using the regular translation method found in KDE, yet you are strongly advised to learn a little on how to translate these (as you will also read in the translator comments).

Please look at http://edu.kde.org/kturtle/translator.php for more information about the translation process. Thanks a lot for your work! KTurtle depends heavily on its translations.

Chapter 7

Credits and License

KTurtle

- Cies Breijs cies AT kde DOT nl
- Anne-Marie Mahfouf annma AT kde DOT org
- Some proofreading changes by Philip Rodrigues phil@kde.org
- Updated translation how-to and some proofreading changes by Andrew Coles andrew_coles AT yahoo DOT co DOT uk

Chapter 8

Index