Chapter 3. Rendering Text

The `pixels` library can do slightly more than just `putPixel`: it also offers a minimal `drawText` proc, to put letters and words on the screen. Its declaration looks like this:

```
proc drawText(x, y: int; text: string; size: int; color: Color)
```

We pass the following parameters to it:

`x` and `y`

The coordinates of a bottom-left corner of the text we want to draw

`text`

The text we want to render. It is of type `string`. We will look at strings in more depth in the next chapter, for now it is enough to know that `string` is a builtin type that is roughly a sequence of `char`, and a *string literal* can be written in double quotes, for example `"like this"`.

`size`

This is a height of the text in pixels.

`color`

Similarly to the `putPixel` proc, we can define the color of the text we're drawing.

The simplest way to call this proc is like this:

```
drawText 30, 40, "Welcome to Nim!", 10, Yellow
```

Notice that we didn't use the parentheses after the name of the function to

enclose the list of arguments. This is equivalent to `drawText(10, 10, "Welcome to Nim!", Yellow)`, either style can be used.

For the next more interesting example we need the *dollar operator* `$` which can turn many types into its string representation, and the *concatenation operator* `&` which combines ("concatenates") two strings into one:

```
$12 == "12"  # convert an integer into a string
"abc" & "def" == "abcdef" # concatenate two strings into one
```

The following example produces 3 lines of text:

```
for i in 1..3:
  let texttodraw = "welcome to nim for the " & $i & "th time!"  ①
  drawtext 10, i*10, texttodraw, 8, Yellow   ②
```

① Creates a string (concatenated from three separate strings) and assigns it to the local variable `textToDraw`.

② Renders the text at position `(10, i*10)` where `i` is in one of the numbers in the `1..3` range, for each loop iteration.

Chapter 4. Sequences

We have said that a string is a sequence of characters. Nim also supports sequences, called `seq`, of an arbitrary type. For example, a sequence of integers is written with the notation `seq[int]`, and a sequence of `Point` is `seq[Point]`.

We want to be able to draw more than a single pixel. `putPixels` accomplishes that:

```
proc putPixels(points: seq[Point]; col: Color) =  ①
  for p in items(points):  ②
    pixels.putPixel p.x, p.y, col ③

putPixels(@[Point(x: 2, y: 3), Point(x: 5, y: 10)], Gold) ④
```

① `putPixels` takes a list of `Point`s.

② The `items` iterator allows us to iterate over the `points` parameter.

③ Every pixel we draw uses the same color `col`. We call the `putPixel` proc from the `pixels` module. As you can see, you can qualify an identifier with the module it was declared in. Sometimes this can improve the readability of your code.

④ We call our newly introduced `putPixels` proc with the `seq @[Point(x: 2, y: 3), Point(x: 5, y: 10)]`.

You can construct a sequence via `@[⋯]`. The empty sequence is `@[]`.

Sequences offer random access, the `i`'ith element can be accessed via `s[i]`. The indexing starts from 0. The same notation is available for `string`.

Chapter 5. Parameter passing and mutability

Every parameter in Nim is *immutable* unless it is declared as a `var` parameter. This means that the following code does not compile:

```
proc resetPointsToOrigin(points: seq[Point]) =
  for i in 0 ..< points.len:      ①
    points[i] = Point(x: 0, y: 0) ②
```

① We iterate over every index of `points` via an iterator that uses an operator symbol `..<`. The `..<` symbol indicates that the upper bound is exclusive which is exactly what we need since the indexing starts at 0.

② We then try to mutate `points[i]` and set its new value to the point `(x: 0, y: 0)`. But the compiler rejects this statement!

The compiler rejects the code because `points` is a parameter that can only be used for read accesses. This restriction helps us to write code that is easier to understand and scales better to larger programs and at the same time it helps the compiler to produce better machine code.

In order to be allowed to mutate `points` we need a `var` parameter:

```
proc resetPointsToOrigin(points: var seq[Point]) = ①
  for i in 0 ..< points.len:
    points[i] = Point(x: 0, y: 0) ②
```

① The `points` parameter is a `var seq`

② so the mutation is allowed.

If we now try to call `resetPointsToOrigin` with a seq constructor the compiler once again rejects our code:

```
resetPointsToOrigin @[Point(x: 2, y: 4)]
```

The reason is that a sequence constructed via `@[]` is not mutable. A variable is mutable, so the following is valid:

```
var points = @[Point(x: 2, y: 4)]
resetPointsToOrigin points
```