

# Splitty's definite Iodine style guide

The first Iodine style guide was written by @Aurora0001. Because he has become inactive and Iodine changed a lot since the last time he updated his style guide, here is a new one, so you can keep writing sexy Iodine code.

I am using this style in my project [iosh](#).

## Let's get started

First things first: Start every module with an Iododoc comment describing the module and listing all authors and contributors.

Here is how your header could look:

```
/**  
 * A description of your module  
 * @module project/src/module  
 * @author You <you@example.com>  
 */
```

The path to the module should be relative to the project root and should not include the file extension ( `.id` ).

## Comments

There are three types of comments in Iodine:

- Single-line comments  
start with a `#` symbol and go till the end of the line.
- Multi-line comments  
start with `/*`, end with `*/` and can go over multiple lines.
- Iododoc comments  
are basically multi-line comments and are used for documentation.

There are no rules for single-line comments. If you find a way to make a single-line comment incredibly ugly by not following some imaginary set of rules, don't hesitate to e-mail me and maybe I'll come up with something to fix that.

For multi-line comments, leave the first line empty, align the content with a space, followed by a `*` symbol and place the `*/` that ends the comment on its own line:

```
/*  
 * Some comment that goes  
 * over multiple lines.  
*/
```

Iododoc comments follow the same rules as multi-line comments.

## Imports


Imports. They are everywhere, so of course there should be a guide on how to aesthetically use them.

Use selective imports if you are including less than five objects:

```
use Collection, Countable, Iterable from std.collections
```

If the line looks aesthetically unpleasing because of its length or you want to import more than five objects, spread the selective imports over multiple lines, aligning them with the first line using four spaces:

```
use Collection, Countable, Iterable,  
    Array, Set, Queue from std.collections
```

As a last resort, if the selective imports span more than three lines or if you actually need most or all types that a specific module offers, use the star (  ) operator to import everything from that module:

```
use * from std.collections
```

## Casing

There are a lot of different types and ways to define a type in Iodine. As a rule of thumb, use UpperCamelCase for everything that can be instantiated, extended or implemented and use snakecase for everything else:

```
class FooBar { ... }  
trait Collection { ... }  
func count () => ...  
func dosomething () => ...
```

# Spaces

Use a lot of spaces, they make stuff readable. You should put a space:

- before accessing an array, tuple or other iterable

```
a [0]
```

- before a parameter list

```
dosomething ()
```

- before and after the arrow (=>) operator

```
lambda () => 0
```

- before and after a binary or logical operator

```
if (a || b && c) { ... }  
a += (b | c)  
(a ^ b) ^ c
```

- before the start and end of a dictionary initializer

```
x = { 'a': 1, 'b': 2 }
```

- before a block following a keyword, an identifier, an assignment or parenthesis

```
for (x in range (100)) { ... }  
lambda (foo, bar) { ... }  
x = { 'a': 1, 'b': 2 }
```

- after a comma, in any situation

```
[1, 2, 3]  
dosomething (1, 2, 3)
```

- after a colon in a parameter list

```
foo (bar: 1, baz: 2)
```

Nice. Now you got a fair amount of spaces in your code.

However, there are also situations where you should not use spaces, here are some of them:

- directly after the start and before the end of a parameter list or list or tuple initializer

```
foo (bar, baz)  
a = [1, 2, 3, 4]  
b = (1, 2, 3, 4)
```

- after a unary operator

```
!foo
```

```
~bar
```

```
-baz
```

# Semicolons

Semicolons are completely optional in Iodine and I suggest not using them at all. However, this is a matter of preference and if you wanna waste your time on typing useless semicolons, there's nothing I can do to stop you ;)

# Examples

People like examples. Here is a `read` function written using this style:

```
/**
 * Reads a line from the standard input stream.
 * @kwparam prompt : Str The prompt
 */
func read (**kwargs) {
    prompt = ""

    /* Handle keyword arguments */
    for (key, value in kwargs) {
        match (key) {
            case "prompt" => prompt = typecast (Str, value)
        }
    }
}
```

```
/* Print the prompt */  
if (!prompt.iswhitespace ()) {  
    stdout.write (prompt)  
}  
  
return stdin.readLine ()  
}
```