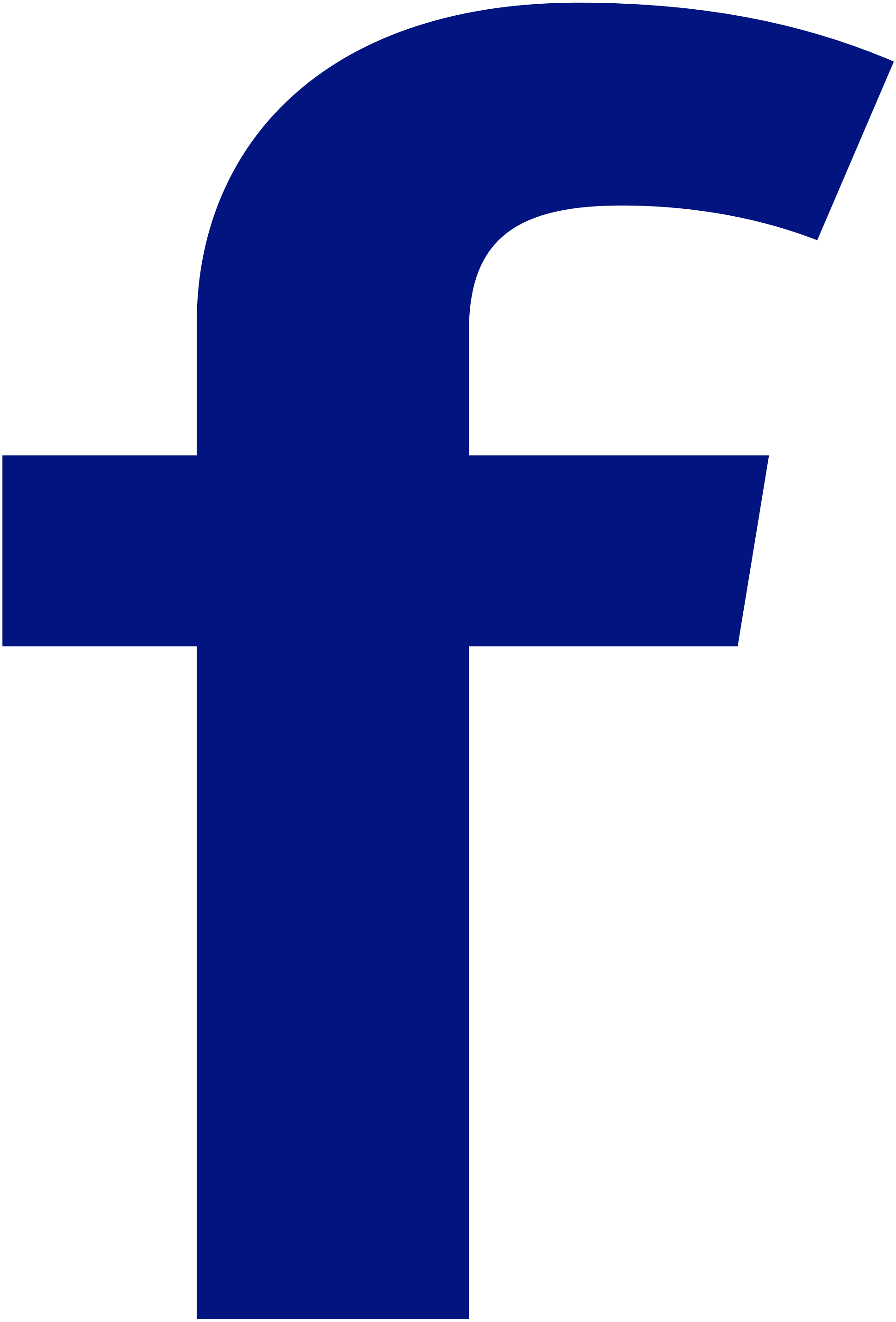
Functional literal with receiver

higher order functions



u

n

a

p

p

ʅ

y

T

W

i

C

e

(

n

●

●

I

n

t

,

f

●

●

I

n

t

●

(

)

[redacted]

>

)

●

●

I

n

t

=

f

(

)

●

f

f

u

n

a

p

p

ʅ

y

T

W

i

C

e

(

n

:

I

n

t

,

f

●

●

(

I

t

)

[redacted]

→

I

n

t

●

●

I

n

t

=

f

(

(

n

)

a

p

p

ʅ

y

T

W

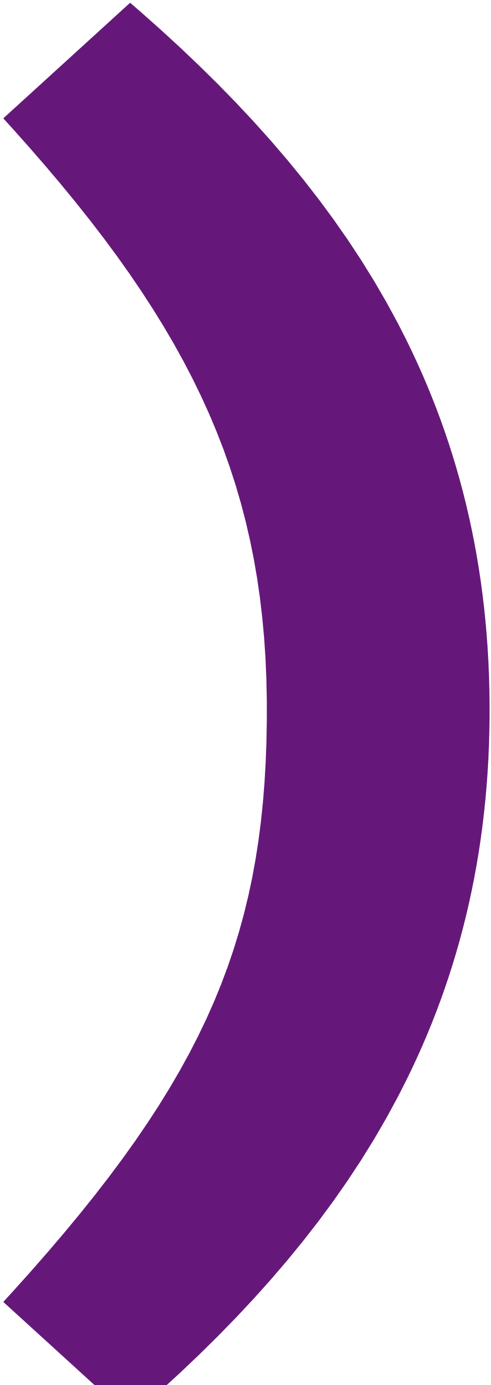
i

C

e

(

2



{

i

t

+

1

}

a

p

p

լ

y

T

W

i

C

e

(

2

)

{

t

h

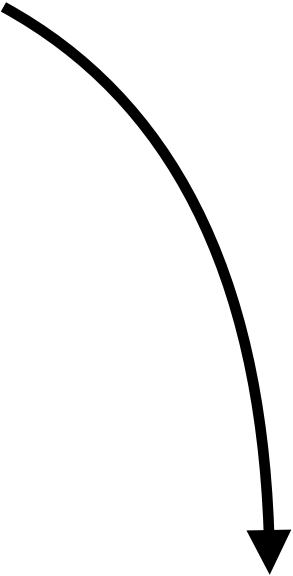
i

S

+

1

}



- `f` can be seen as a local extension function on the receiver type
- the receiver type is `Int`
- the lambda has no param, the param becomes **`this`**
- the receiver object (`this`) is `2` on the first call and the result of `2.f()` (3) on the second call

```kotlin
fun applyTwice(n: Int, f: (Int) -> Int): Int = f(f(n))
```

```
applyTwice(2) { it + 1 }
```

```kotlin
fun applyTwice(n: Int, f: Int.() -> Int): Int = n.f().f()
```

```
applyTwice(2) { this + 1 }
```

Functional literal with receiver

higher order functions

```
fun applyTwice(n: Int, f: (Int) → Int): Int = f(f(n))

applyTwice(2) { it + 1 }
```

```
fun applyTwice(n: Int, f: Int.() -> Int): Int = n.f().f()

applyTwice(2) { this + 1 }
```

- f can be seen as a local extension function on the receiver type
- the receiver type is `Int`
- the lambda has no param, the param becomes **`this`**
- the receiver object (`this`) is 2 on the first call and the result of `2.f()` (3) on the second call

Operator overloading