They are compiled into a class that is instantiated at runtime as a function value.

It's implemented through the partially-applied function:

```
nums.foreach(println _)
```

The concise form only works where a compatible function is expected.

Put `*` after the type of the repeated parameter:

```
def echo(args: string*)=//...
```

Available as an `Array` of the parameter type.

Function literals are *closed* terms, since they have no free variables.

Closures are *open* terms (refer to variables defined elsewhere), and close the function by capturing the binding of the free variables.

It's an instance of a class that extends one of several `FunctionN` traits in the package scala.

Inner classes can't access modifiable variables and surrounding scopes, so there's no difference between capturing a variable and capturing its current value.

Scala captures variables so it "sees" changes.

Put `: _*` after name

```
echo(arr: _*)
```

e.g.,

`def echo (args String*)
  for (arg <- args) println(arg)`

They show only one call of the recursive function, unless you use

```
-g:notailcalls
```

to scala shell or to `scalac`.

It will capture the one currently active. The closure keeps the variable alive after the method returns, automatically moving it from the stack to the heap, if necessary.

It avoids situations where the programmer forgets to provide the right parameters, giving the unexpected result of a *function value* instead of the application of a function.

Hence _ can be left off **only** when a function is expected.

Use a partially-applied function.

```
def sum(a:Int, b:Int, c:Int) = a*b*c
val a = sum _
a(1,2,3)
```

They represent one arg each, not multiple references to the same one.

It's only for methods and local functions without intermediaries.

It will not work with function values, or intermediaries such as one method tail-calling another method that calls it back.

Source code vs. objects at runtime.

Use curly braces.

There's no need to pass outer functions' args.