How are function literals implemented?

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

How is `nums.foreach(println)` implemented?

It's implemented through the partially-applied function:

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

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

Describe repeated parameter syntax.

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

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

Available as an `Array` of the parameter type.

What makes a function literal a *closure*?

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.

What is a function value?

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

Contrast closures and Java inner classes.

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.

Pass array to a repeated parameter.

Put `: _*` after name

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

e.g.,
`def echo (args String*)
  for (arg <- args) println(arg)`

Describe tail-recursive stack traces.

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

```
 -g:notailcalls
```

to scala shell or to `scalac`.

Which instance of a local variable of a method, repeatedly called, will a closure capture?

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.

Why the trailing underscore?

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.

Assign a method or local function to a variable.

Use a partially-applied function.

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

What do multiple underscores in a function literal do?

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

What are the limits of tail recursion?

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.

Function literals vs. function values

Source code vs. objects at runtime.

What is the syntax for multiline function literals?

Use curly braces.

Why do local functions reduce boilerplate over private methods?

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