JN2525-Classes

Accessing Private Variables

Closures and functions can't remember any information defined within themselves between invocations. If we want a closure to remember a variable between invocations, one only it has access to, we can nest the definitions inside a block:

```
def c
try{
  def a= new Random() //only closure c can see this variable; it is private to c
  c= { a.nextInt(100) }
}
100.times{ println c() }
try{ a; assert 0 }catch(e) //'a' inaccessable here
{ assert e instanceof MissingPropertyException }
```

We can have more than one closure accessing this private variable:

```
def counterInit, counterIncr, counterDecr, counterShow
      //common beginning of names to show common private variable/s
try{
  def count
  counterInit= { count= it }
  counterIncr= { count++ }
  counterDecr= { count-- }
  counterShow= { count }
}
counterInit(0)
counterIncr(); counterIncr(); counterDecr(); counterIncr()
assert counterShow() == 2
```

We can also put all closures accessing common private variables in a map to show they're related:

```
def counter= [:]
try{
  def count= 0
  counter.incr= { count++; counter.show() }
  counter.decr= { count--; counter.show() }
  counter.show= { count }
}

counter.incr()
assert counter.show() == 1
```

Expando

We can access private variables with an Expando instead. An expando allows us to assign closures to Expando names:

```
def counter= new Expando()
try{
  def count= 0
  counter.incr= { count++; show() }
           //no need to qualify closure call with expando name
  counter.decr= { count--; show() }
  counter.show= { timesShown++; count }
  counter.timesShown= 0
           //we can associate any value, not just closures, to expando keys
}
counter.incr(); counter.incr(); counter.decr(); counter.incr()
assert counter.show() == 2
```

An expando can also be used when common private variables aren't used:

```
def language= new Expando()
language.name= "Groovy"
language.numLetters= { name.size() }

assert language.numLetters() == 6
language.name= "Ruby"
assert language.numLetters() == 4
language.name= "PHP"
assert language.numLetters() == 3
```

Like individual closures, closures in expandos see all external variables all the way to the outermost block. This is not always helpful for large programs as it can limit our choice of names:

```
def a= 7
try{
  //... ... ... lots of lines and blocks in between ... ... ...
  def exp= new Expando()
  exp.c= {
    //def a= 2 //does not compile if uncommented: a is already defined
    //... ... ...
  }
}
```

For single-argument closures, both standalone and within expandos, we can use the implicit parameter as a map for all variables to ensure they're all valid, though the syntax is not very elegant:

```
def a= 7
try{
  def c= {
    it= [it: it]
    it.a= 2
    it.it + it.a
 }
  assert c(3) == 5
}
```

There is a better way to ensure a chosen variable name will not "shadow" another from the same scope.

Static Classes

Just as we can use functions instead of closures to hide names from the surrounding context, so also we can use static classes instead of expandos to hide such external names. We use the static keyword to qualify the individual definitions in a class definition:

```
def a= 7
def a= 7
class Counter{
  //variable within a class is called a field...
  static public count= 0
      //count has 'public' keyword, meaning it's visible from outside class

  //function within a class is called a method...
  static incr(){
    count++
      //variables defined within class visible from everywhere else inside class
  }
  static decr(){
    //println a //compile error if uncommented:
                //a is outside the class and not visible
    count--
  }
}
Counter.incr(); Counter.incr(); Counter.decr(); 5.times{ Counter.incr() }
assert Counter.count == 6
```

Methods act quite similar to standalone functions. They can take parameters:

```
class Counter{
  static private count = 0
      //qualified with private, meaning not visible from outside class
  static incr( n ){ count += n }
  static decr( count ){ this.count -= count }
      //params can have same name as a field; 'this.' prefix accesses field
  static show(){ count }
}
Counter.incr(2); Counter.incr(7); Counter.decr(4); Counter.incr(6)
assert Counter.show() == 11
```

We can have more than one method of the same name if they each have different numbers of parameters.

```
class Counter{
  static private count = 0
  static incr(){ count++ }
  static incr( n ){ count += n }
  static decr(){ count-- }
  static decr( n ){ count -= n }
  static show(){ count }
}
Counter.incr(17); Counter.incr(); Counter.decr(4)
assert Counter.show() == 14
```

Methods are also similar to other aspects of functions:

```
class U{
  static a(x, Closure c){ c(x) }
  static b( a, b=2 ){ "$a, $b" } //last argument/s assigned default values
  static c( arg, Object[] extras ){ arg + extras.inject(0){ flo, it-> flo+it } }
  static gcd( m, n ){ if( m%n == 0 )return n; gcd(n,m%n) }
                                          //recursion by calling own name
}
assert U.a(7){ it*it } == 49 //shorthand passing of closures as parameters
assert U.b(7, 4) == '7, 4'
assert U.b(9) == '9, 2'
assert U.c(1,2,3,4,5) == 15 //varying number of arguments using Object[]
assert U.gcd( 28, 35 ) == 7
```

We can assign each method of a static class to a variable and access it directly similar to how we can with functions:

```
class U{
  static private a= 11
  static f(n){ a*n }
}
assert U.f(4) == 44
def g= U.&f //special syntax to assign method to variable
assert g(4) == 44
def h = g //don't use special syntax here
assert h(4) == 44
```

When there's no accessibility keyword like 'public' or 'private' in front of a field within a static class, it becomes a property, meaning two extra methods are created:

```
class Counter{
  static count = 0
       //property because no accessibility keyword (eg 'public','private')
  static incr( n ){ count += n }
  static decr( n ){ count -= n }
}
Counter.incr(7); Counter.decr(4)
assert Counter.count == 3
assert Counter.getCount() == 3 //extra method for property, called a 'getter'
Counter.setCount(34)           //extra method for property, called a 'setter'
assert Counter.getCount() == 34
```

When we access the property value using normal syntax, the 'getter' or 'setter' is also called:

```
class Counter{
  static count= 0 //'count' is a property

  //we can define our own logic for the getter and/or setter...
  static setCount(n){ count= n*2 } //set the value to twice what's supplied
  static getCount(){ 'count: '+ count }
                   //return the value as a String with 'count: ' prepended
}
Counter.setCount(23) //our own 'setCount' method is called here
assert Counter.getCount() == 'count: 46'
                     //our own 'getCount' method is called here
assert Counter.count == 'count: 46'
                     //our own 'getCount' method is also called here
Counter.count= 7
assert Counter.count == 'count: 14'
                 //our own 'setCount' method was also called in previous line
```

To run some code, called a static initializer, the first time the static class is accessed. We can have more than one static initializer in a class.

```
class Counter{
  static count = 0
  static{ println 'Counter first accessed' } //static initializer
  static incr( n ){ count += n }
  static decr( n ){ count -= n }
}
println 'incrementing...'
Counter.incr(7) //'Counter first accessed' printed here
println 'decrementing...'
Counter.decr(4) //nothing printed
```

Instantiable Classes

We can write instantiable classes, templates from which we can construct many instances, called objects or class instances. We don't use the static keyword before the definitions within the class:

```
class Counter{
  def count = 0 //must use def inside classes if no other keyword before name
  def incr( n ){ count += n }
  def decr( n ){ count -= n }
}
def c1= new Counter() //create a new object from class
c1.incr(2); c1.incr(7); c1.decr(4); c1.incr(6)
assert c1.count == 11

def c2= new Counter() //create another new object from class
c2.incr(5); c2.decr(2)
assert c2.count == 3
```

We can run some code the first time each object instance is constructed. First, the instance initializer/s are run. Next run is the constructor with the same number of arguments as in the calling code.

```
class Counter{
  def count
  { println 'Counter created' }
      //instance initializer shown by using standalone curlies
  Counter(){ count= 0 }
      //instance constructor shown by using class name
  Counter(n){ count= n }
      //another constructor with a different number of arguments
  def incr( n ){ count += n }
  def decr( n ){ count -= n }
}
c = new Counter() //'Counter created' printed
c.incr(17); c.decr(2)
assert c.count == 15
d = new Counter(2) //'Counter created' printed again
d.incr(12); d.decr(10); d.incr(3)
assert d.count == 7
```

If we don't define any constructors, we can pass values directly to fields within a class by adding them to the constructor call:

```
class Dog{
  def sit
  def number
  def train(){ ([sit()] * number).join(' ') }
}
def d= new Dog( number:3, sit:{'Down boy!'} )
assert d.train() == 'Down boy! Down boy! Down boy!'
```

Methods, properties, and fields on instantiable classes act similarly to those on static classes:

```
class U{
  private timesCalled= 0 //qualified with visibility, therefore a field
  def count = 0 //a property
  def a(x){ x }
  def a(x, Closure c){ c(x) } //more than one method of the same name but
                              //each having different numbers of parameters

  def b( a, b=2 ){ "$a, $b" } //last argument/s assigned default values
  def c( arg, Object[] extras ){ arg + extras.inject(0){ flo, it-> flo+it } }
  def gcd( m, n ){ if( m%n == 0 )return n; gcd(n,m%n) }
                              //recursion by calling own name
}
def u=new U()
assert u.a(7){ it*it } == 49 //shorthand passing of closures as parameters
assert u.b(7, 4) == '7, 4'
assert u.b(9) == '9, 2'
assert u.c(1,2,3,4,5) == 15 //varying number of arguments using Object[]
assert u.gcd( 28, 35 ) == 7
u.setCount(91)
assert u.getCount() == 91
```

A class can have both static and instantiable parts by using the static keyword on the definitions that are static and not using it on those that are instantiable:

```
class Dice{
  //here is the static portion of the class...
  static private count //doesn't need a value
  static{ println 'First use'; count = 0 }
  static showCount(){ return count }

  //and here is the instantiable portion...
  def lastThrow
  Dice(){ println 'Instance created'; count++ }

  //static portion can be used by instantiable portion, but not vice versa
  def throww(){
    lastThrow = 1+Math.round(6*Math.random()) //random integer from 1 to 6
    return lastThrow
  }
}
d1 = new Dice() //'First use' then 'Instance created' printed
d2 = new Dice() //'Instance created' printed
println "Dice 1: ${(1..20).collect{d1.throww()}}"
println "Dice 2: ${(1..20).collect{d2.throww()}}"
println "Dice 1 last throw: $d1.lastThrow, dice 2 last throw: $d2.lastThrow"
println "Number of dice in play: ${Dice.showCount()}"
```

A class can have more than one constructor:

```
class A{
  def list= []
  A(){
    list<< "A constructed"
  }
  A(int i){
    this()
      //a constructor can call another constructor if it's the first statement
    list<< "A constructed with $i"
  }
  A(String s){
    this(5)
    list<< "A constructed with '$s'"
  }
}
def a1= new A()
assert a1.list == [ "A constructed" ]

def a2= new A(7)
assert a2.list.collect{it as String} == [
  "A constructed",
  "A constructed with 7",
]

def a3= new A('bird')
assert a3.list.collect{it as String} == [
  "A constructed",
  "A constructed with 5",
  "A constructed with 'bird'",
]
```

Categories

When a class has a category method, that is, a static method where the first parameter acts like an instance of the class, we can use an alternative 'category' syntax to call that method:

```
class View{
  def zoom= 1
  def produce(str){ str*zoom }
  static swap(self, that){ //first parameter acts like instance of the class
    def a= self.zoom
    self.zoom= that.zoom
    that.zoom= a
  }
}
def v1= new View(zoom: 5), v2= new View(zoom: 4)
View.swap( v1, v2 ) //usual syntax
assert v1.zoom == 4 && v2.zoom == 5
use(View){ v1.swap( v2 ) } //alternative syntax
assert v1.zoom == 5 && v2.zoom == 4
assert v1.produce('a') == 'aaaaa'
```

We can also use category syntax when the category method/s are in a different class:

```
class View{
  static timesCalled= 0 //unrelated static definition
  def zoom= 1
  def produce(str){ timesCalled++; str*zoom }
}
class Extra{
  static swap(self, that){ //first parameter acts like instance of View class
    def a= self.zoom
    self.zoom= that.zoom
    that.zoom= a
  }
}
def v1= new View(zoom: 5), v2= new View(zoom: 4)
use(Extra){ v1.swap( v2 ) }
      //alternative syntax with category method in different class
assert v1.zoom == 4 && v2.zoom == 5
assert v1.produce('a') == 'aaaa'
```

Many supplied library classes in Groovy have category methods that can be called using category syntax. (However, most category methods on Numbers, Characters, and Booleans do not work with category syntax in Groovy-1.0)

```
assert String.format('Hello, %1$s.', 42) == 'Hello, 42.'
use(String){
  assert 'Hello, %1$s.'.format(42) == 'Hello, 42.'
}
```

Far more common are supplied library classes having category methods in another utility class, eg, List having utilities in Collections:

```
def list= ['a', 7, 'b', 9, 7, 7, 2.4, 7]
Collections.replaceAll( list, 7, 55 ) //normal syntax
assert list == ['a', 55, 'b', 9, 55, 55, 2.4, 55]

list= ['a', 7, 'b', 9, 7, 7, 2.4, 7]
use(Collections){
  list.replaceAll(7, 55) //category syntax
}
assert list == ['a', 55, 'b', 9, 55, 55, 2.4, 55]
```

We can call category methods inside other category methods:

```
class Extras{
  static f(self, n){ "Hello, $n" }
}
class Extras2{
  static g(self, n){
    Extras.f(self, n)
  }
  static h(self, n){
    def ret
    use(Extras){ ret= self.f(n) } //call Extras.f() as a category method
    ret
  }
}
assert Extras.f(new Extras(), 4) == 'Hello, 4'
assert Extras2.g(new Extras2(), 5) == 'Hello, 5'
assert Extras2.h(new Extras2(), 6) == 'Hello, 6'

class A{ }
def a= new A()
use(Extras){
  assert a.f(14) == 'Hello, 14'
}
use(Extras2){
  assert a.g(15) == 'Hello, 15'
  assert a.h(16) == 'Hello, 16' //call category method within another
}
```

But we can't call category methods inside another category method from the same class:

```
class Extras{
  static f(self, n){ "Hello, $n" }
  static g(self, n){ f(self, n) }
  static h1(self, n){ f(n) } //calling f without first parameter only valid
                             //when called within a category method
  static h2(self, n){
    def ret
    use(Extras){
      ret= self.f(n)
    } //class as category within itself only valid if method wasn't called
                                                          //using category syntax
    ret
  }
}
assert Extras.f(new Extras(), 4) == 'Hello, 4'
assert Extras.g(new Extras(), 5) == 'Hello, 5'
try{ Extras.h1(new Extras(), 6); assert 0 }
catch(e){ assert e instanceof MissingMethodException }
assert Extras.h2(new Extras(), 7) == 'Hello, 7'

class A{ }
def a= new A()
use(Extras){
  assert a.f(14) == 'Hello, 14'
  assert a.g(15) == 'Hello, 15'
  assert a.h1(16) == 'Hello, 16'
  try{ a.h2(17); assert 0 }
  catch(e){ assert e instanceof GroovyRuntimeException }
}
```

A lot of entities in Groovy are classes, not just the explicit ones we've just learnt about. Numbers, lists, sets, maps, strings, patterns, scripts, closures, functions, and expandos are all implemented under the hood as classes. Classes are the building block of Groovy.