Lecture: S4 classes and methods

Martin Morgan, Robert Gentleman
Fred Hutchinson Cancer Research Center
Seattle, WA, USA

14 February, 2008

Object oriented programming

class	Data encapsulation
method	Set / get, show, transformation
inheritance	For data and method reuse

Flavors

S3 Convenient, *ad hoc*, single inheritance, single dispatch, instance-based.

S4 Formal, multiple inheritance & dispatch. Introspection.

Benefits

- Abstract data types – interface to data.
- Reuse – data components (e.g., experiment description), inheritance (e.g., *Sequences* vs. *DNASequences*)

Examples: S3

```
> example(lm)

> class(lm.D90)

[1] "lm"

> head(names(lm.D90), n = 4)

[1] "coefficients" "residuals"
[3] "effects"      "rank"

> head(methods("summary"), n = 4)

[1] "summary.aov"        "summary.aovlist"
[3] "summary.connection" "summary.data.frame"

> head(methods(class = "lm"), n = 4)

[1] "add1.lm"       "alias.lm"
[3] "anova.lm"      "case.names.lm"
```

Examples: S4 I

```
> library(Biobase)
> data(sample.ExpressionSet)
> class(sample.ExpressionSet)

[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
```

Examples: S4 II

```
> getClass("ExpressionSet")

Slots:

Name:            assayData          phenoData
Class:           AssayData AnnotatedDataFrame

Name:          featureData     experimentData
Class:  AnnotatedDataFrame              MIAME

Name:           annotation  .__classVersion__
Class:           character           Versions

Extends:
Class "eSet", directly
Class "VersionedBiobase", by class "eSet", distance 2
Class "Versioned", by class "eSet", distance 3
```

Examples: S4 III

Class introspection

- `getClass`
- `getSlots`, `slotNames`
- `extends`

Method introspection

- `showMethods("exprs")`, `showMethods(class="ExpressionSet")`
- `getMethod("exprs", "ExpressionSet")`

Class definition: `setClass`

```
> setClass("Sequences",
+          representation=representation(
+            sequences="character"))
> setClass("DNASequences",
+          contains="Sequences",
+          representation=representation(
+            chromosome="character"))
```

Class definition

```
> names(formals(setClass))
 [1] "Class"            "representation"
 [3] "prototype"        "contains"
 [5] "validity"         "access"
 [7] "where"            "version"
 [9] "sealed"           "package"
```

- *representation*: named list of ‘slots’ and their classes
- *prototype*: named list of slots and default (e.g., `character(0)` values
- *contains*: character vector of contained (inheritted) classes
- *validity*: programatic constraints on object contents; also `setValidity`

Validity

```
> setValidity("DNASequences", function(object) {
+   msg <- NULL
+   atgc <- grep("[^atcg]", sequences(object))
+   if (length(atgc)>0)
+     msg <- c(msg, "'sequences' must be a, t, c, or g")
+   if (is.null(msg)) TRUE
+   else msg
+ })
```

- Implicitly called when object instantiated.
- Explicit usage: `validObject(dnaSeq)`

Validity

- Impose constraints beyond type
- Argument to `setClass`, or call to `setValidity`
- Function returns `TRUE` or character string describing invalid aspect(s)
- Automatically called during object creation, or with `validObjecct`
- (Advanced) Special dispatch: do *not* `callNextMethod`; check *only* validity of specific class

Slot access

- Get: `@`, `slot`
- Set: `@<-`, `slot<-`

Usually, use *accessor* methods instead

- Goal: separate interface from implementation.
- ‘Getters’ for all (publically accessible) slots
- ‘Setters’ for slots intended to be mutable

Get: a simple method

```
> setGeneric("sequences",
+            function(object) {
+              standardGeneric("sequences")
+            })
> setMethod("sequences",
+           signature(object="Sequences"),
+           function(object) {
+             slot(object, "sequences")
+           })
```

Usage: `sequences(dnaSeq)`

Set: a replacement method

```
> setGeneric("sequences<-",
+            function(object, value) {
+                standardGeneric("sequences<-")
+            })
> setReplaceMethod("sequences",
+                  signature(object="Sequences"),
+                  function(object, value) {
+                    slot(object, "sequences") <-
+                      tolower(value)
+                    validObject(object)
+                    object
+                  })
```

Usage: `sequences(dnaSeq) <- "aaaccttt"`

Defining generics

```
> names(formals(setGeneric))
```

```
[1] "name"              "def"
[3] "group"             "valueClass"
[5] "where"             "package"
[7] "signature"         "useAsDefault"
[9] "genericFunction"
```

name Name of an existing function (to be used as the default) or a new name.

def Function definition with named argumments and defintion. *def* Used for *dispatch* rather than evaluation; body usually `standardGeneric(<name>)`.

signature Character vector of named arugments to be used for dispatch (some details below).

Defining methods

```
> names(formals(setMethod))
[1] "f"          "signature"  "definition"
[4] "where"      "valueClass" "sealed"
```

f	Name of the generic
signature	Named character vector matching argument names to types. Implicit type is `ANY`, another type is `missing`
definition	function definition, matching generic.

Reuse and inheritance: `show`

```
> setMethod("show",
+           signature=signature(
+             object="Sequences"),
+           function(object) {
+             cat("class:", class(object), "\n")
+             cat("sequences:", sequences(object), "\n")
+           })
> setMethod("show",
+           signature=signature(
+             object="DNASequences"),
+           function(object) {
+             callNextMethod()
+             cat("chromosome:", chromosome(object), "\n")
+           })
```

Instantiation: new

```
> dnaSeq <- new("DNASequences", sequences = "aatat",
+     chromosome = "X")
> dnaSeq
```

```
class: DNASequences
sequences: aatat
chromosome: X
```

initialize

```
> setMethod("initialize",
+           signature(.Object="Sequences"),
+           function(.Object, ..., sequences=character(0)){
+               sequences <- tolower(sequences)
+               callNextMethod(.Object, ...,
+                              sequences=sequences)
+           })
> new("DNASequences", sequences = "AATAT",
+     chromosome = "X")
class: DNASequences
sequences: aatat
chromosome: X
```

Instantiation

```
> names(formals(new))
[1] "Class" "..."
```

- Typically: *...* at most one unnamed element (e.g., *.Data*, used to initialize super class) and additional named arguments (names often correspond to slots).
- The class *prototype* is used as a template, updated by named arguments

Mutliple inheritance, virtual classes

- Multiple inheritance: several *contains* classes
- Virtual classes: group related data types
- `setClassUnion`: establish ‘extends’ relationships between existing classes

Multiple inheritance and class unions I

```
> setClass("A",
+            representation = representation(
+              x="numeric"))

[1] "A"

> setClass("B",
+            representation = representation(
+              y="numeric"))

[1] "B"

> setClass("AB",
+            contains=c("A", "B"))

[1] "AB"
```

Multiple inheritance and class unions II

```
> new("AB")

An object of class "AB"
Slot "x":
numeric(0)

Slot "y":
numeric(0)
```

setClassUnion I

```
> setClassUnion("AOrB", c("A", "B"))

[1] "AOrB"

> getClass("AOrB")

Extended class definition ( "ClassUnionRepresentation" )
Virtual Class

No Slots, prototype of class "NULL"

Known Subclasses:
Class "A", directly
Class "B", directly
Class "AB", by class "A", distance 2
Class "AB", by class "B", distance 2
```

setClassUnion II

```
> getClass("A")
Slots:

Name:          x
Class:  numeric

Extends: "AOrB"

Known Subclasses: "AB"
```

- *A* now extends *AOrB*!

Real example: class union

```
> getClass("AssayData")

Extended class definition ( "ClassUnionRepresentation" )
Virtual Class

No Slots, prototype of class "NULL"

Known Subclasses:
Class "list", directly
Class "environment", directly
Class "Versions", by class "list", distance 2
Class "VersionsNull", by class "list", distance 3
```

Dispatch and inheritance

- *Multiple dispatch* when more than one argument in signature, e.g., `"["`
- Dispatch to first matching signature in linearized method list
- 'Matching' signature: compare class of supplied object(s) with classes names in method definition.
- Possibly several signatures match:
  - Inheritance (e.g., B extends A; method *foo* for classes A, B; argument is instance of B; both *foo* possible)
  - Multiple arugments, some with signature `ANY`
  - Both inheritance and multiple arguments
  - Methods ordered in terms of 'distance' from suplied arguments; complex method lists lead to (very) complex distance calculations
- `callNextMethod` calls 'next' method in linearized method list.

S4 and packages

DESCIPTION

- Depends: `methods`
- Imports: other package classes and methods

NAMESPACE

- `importClassesFrom`
- `import`: usually generics or regular functions
- `exportClasses`
- `export`: including generics
- `exportMethods`: for methods on generics defined in other packages, e.g., `show`, `initialize`

Documentation

- `promptClass`, `promptMethods`

`new` and `initialize` I

MTM: Implicitly:

- `new("Sequences")` must work (used during validity checking).
- `new("DNASequences", obj, chromosome="Y")` is a *copy constructor*, using `obj` as a template for creating a new `DNASequences` object.
- `callNextMethod()` should work, without special effort, for derived classes.

Consequently. . .

new and initialize II

```
> setMethod("initialize",
+           signature=signature(
+               .Object="DNASequences"),
+           function(.Object, ..., sequences=character(0))
+               sequences <- toupper(sequences)
+               callNextMethod(.Object, ...,
+                              sequences=sequences)
+      })
[1] "initialize"
```

- Only slot names as argument to *initialize* methods.
- Only include arguments for slots defined in the class for which *initialize* is specialized to.
- Force arguments to *initialize* to be named.

Constructors I

MTM: `new` is a 'low-level' function, suitable for class authors but perhaps not the users.

- Exposes class structure, breaking the abstraction layer.
- Restricts arguments to slot names.
- Provides no hints to user about appropriate arguments.
- Requires class author and user to employ same methods for object creation.

Constructors II

Solution

```
> DNASequences <- function(uri, format = "fasta",
+     ...) {
+     sequences <- paste(readLines(uri)[-1],
+         collaspe = "")
+     new("DNASequences", sequences = sequences,
+         ...)
+ }
```

- *initialize* does not need to be exported
- Constructor can be a generic, with methods.

Creating accessors programatically

- Getters and setters are very standardized.
- Makes sense to write a function `.accessors` to create appropriate generics and methods (see `GSEABase:::.accessors` for an example)

Example: getters and setters created with

```
> GSEABase:::.accessors("Sequences")
> GSEABase:::.accessors("DNASequences")
```