Object-Oriented Programming

Kurt Hornik

Outline

- Motivation
- S3 Classes
- S4 Classes
- Reference Classes

Needs in data analysis

- Support structural thinking (numeric vectors, factors, data frames, results of fitting a model, . . .)

Needs in data analysis

- Support structural thinking (numeric vectors, factors, data frames, results of fitting a model, . . .)
- Support functional thinking (“generic” functions)

Needs in data analysis

- Support structural thinking (numeric vectors, factors, data frames, results of fitting a model, . . .)
- Support functional thinking (“generic” functions)
- Allow for simple extensibility (printing, plotting, summarizing, . . .)

Outline

- Motivation
- S3 Classes
- S4 Classes
- Reference Classes

Basic idea

- Specify the class via the class attribute

Basic idea

- Specify the class via the class attribute
- Have a way of making a function “generic”

Basic idea

- Specify the class via the class attribute
- Have a way of making a function “generic”
- The “method” of generic `foo` for class `bar` is called `foo.bar`

Basic idea

- Specify the class via the class attribute
- Have a way of making a function “generic”
- The “method” of generic `foo` for class `bar` is called `foo.bar`
- Allow for default methods called `foo.default`

Outline

- Motivation
- S3 Classes
  - Classes
  - Generic and Method Functions
  - Group Methods
  - Odds and ends
- S4 Classes
- Reference Classes

Manipulating the class info

- Function `class()` gets the class attribute, and there is a replacement function for setting it:

```
R> x <- 1 : 3
R> x
[1] 1 2 3
```

Manipulating the class info

- Function `class()` gets the class attribute, and there is a replacement function for setting it:

```
R> x <- 1 : 3
R> x
[1] 1 2 3
R> class(x) <- "bar"
R> x
[1] 1 2 3
attr(,"class")
[1] "bar"
```

Manipulating the class info

- Function `class()` gets the class attribute, and there is a replacement function for setting it:

```
R> x <- 1 : 3
R> x
[1] 1 2 3
R> class(x) <- "bar"
R> x
[1] 1 2 3
attr(,"class")
[1] "bar"
```

- If e.g. `class(x)` is `c("bar", "baz", "grr")`, then x belongs to class `bar` and inherits from `baz`, then `grr`, etc.

Manipulating the class info

- Function `class()` gets the class attribute, and there is a replacement function for setting it:

```
R> x <- 1 : 3
R> x

[1] 1 2 3

R> class(x) <- "bar"
R> x

[1] 1 2 3
attr(,"class")
[1] "bar"
```

- If e.g. `class(x)` is `c("bar", "baz", "grr")`, then x belongs to class `bar` and inherits from `baz`, then `grr`, etc.
- Function `unclass()` removes the class(es)

Outline

- Motivation
- S3 Classes
  - Classes
  - Generic and Method Functions
  - Group Methods
  - Odds and ends
- S4 Classes
- Reference Classes

Method dispatch

- A generic function tries to find methods for its primary argument

Method dispatch

- A generic function tries to find methods for its primary argument
- First, using the *GENERIC.CLASS* naming convention for each class the primary arg inherits from

Method dispatch

- A generic function tries to find methods for its primary argument
- First, using the *GENERIC.CLASS* naming convention for each class the primary arg inherits from
- If no such method found, try *GENERIC*`.default` (default method)

Method dispatch

- A generic function tries to find methods for its primary argument
- First, using the *`GENERIC.CLASS`* naming convention for each class the primary arg inherits from
- If no such method found, try *`GENERIC`*`.default` (default method)
- If not found either: error.

Creating generics

Typically via `UseMethod()`. E.g.,

```
R> foo <- function(x, ...) UseMethod("foo")
R> foo

function(x, ...) UseMethod("foo")
```

Creating methods

Simply create functions obeying the *GENERIC.CLASS* naming convention. E.g.,

```
R> foo.bar <- function(x, ...) cat("I am here\n")
R> foo.bar

function(x, ...) cat("I am here\n")
```

Basic dispatch by calling the generic:

```
R> foo(x)

I am here
```

Explicitly invoking inheritance

Function `NextMethod()` can be used within methods to call the next method. E.g.,

```
R> test <- function(x) UseMethod("test")
R> test.c1 <- function(x) { cat("c1\n"); NextMethod(); x }
R> test.c2 <- function(x) { cat("c2\n"); NextMethod(); x }
R> test.c3 <- function(x) { cat("c3\n"); NextMethod(); x }
R> x <- 1
R> class(x) <- c("c1", "c2")
```

What will happen?

```
R> test(x)
```

Dispatching on other arguments

Specify argument for dispatch in the `UseMethod()` call. E.g.,

```
foo <- function(x, y, ...) UseMethod("foo", y)
```

Testing and coercion

No “formal” testing (predicate) for and coercion to S3 classes. By convention, `is.bar()` and `as.bar()`. E.g.,

```
R> is.bar <- function(x) inherits(x, "bar")
R> as.bar <- function(x) if(is.bar(x)) x else bar(x)
```

(assuming that `bar()` creates objects of class `"bar"`).

Outline

- Motivation
- S3 Classes
  - Classes
  - Generic and Method Functions
  - Group Methods
  - Odds and ends
- S4 Classes
- Reference Classes

Motivation

Issues:

- How can we e.g. add or compare objects of a certain class?
- Or more generally, e.g. compare with objects not in the same class?

Need a dispatch mechanism for “operators” (such as "<") which works in *both* arguments!

Group methods

Mechanism:

- Operators grouped in three categories (Math, Ops, Summary)

Group methods

Mechanism:

- Operators grouped in three categories (Math, Ops, Summary)
- Invoked if the operands correspond to the same method, or one to a method that takes precedence; otherwise, the default method is used.

Group methods

Mechanism:

- Operators grouped in three categories (Math, Ops, Summary)
- Invoked if the operands correspond to the same method, or one to a method that takes precedence; otherwise, the default method is used.
- Class methods dominate group methods.

Group methods

Mechanism:

- Operators grouped in three categories (Math, Ops, Summary)
- Invoked if the operands correspond to the same method, or one to a method that takes precedence; otherwise, the default method is used.
- Class methods dominate group methods.
- Dispatch info available: `.Method`, `.Generic`, `.Group`, `.Class`

Existing group methods

Math	abs sign sqrt floor ceiling trunc round signif exp log cos sin tan acos asin atan cosh sinh tanh acosh asinh atanh lgamma gamma gammaCody digamma trigamma tetragamma pentagamma cumsum cumprod cummax cummin
Ops	+ - * / ^ %% %/% & | ! == != < <= >= >
Summary	all any sum prod min max range

Outline

- Motivation
- S3 Classes
  - Classes
  - Generic and Method Functions
  - Group Methods
  - Odds and ends
- S4 Classes
- Reference Classes

Internal generics

In addition to UseMethod dispatch and group methods, some functions dispatch “internally” (`DispatchOrEval()` in the underlying C code of builtin functions):

- subscripting and subassigning (`[`, `[[`, `$`)
- length dim dimnames c unlist
- as.character as.vector
- many is.xxx functions for builtins data types xxx

Strengths and weaknesses

- Simple and powerful as long as the naming convention is adhered to

Strengths and weaknesses

- Simple and powerful as long as the naming convention is adhered to
- No formal class structure (can have objects with class `c("foo", "U")` and `c("foo", "V")`, no structural integrity, ...)

Strengths and weaknesses

- Simple and powerful as long as the naming convention is adhered to
- No formal class structure (can have objects with class `c("foo", "U")` and `c("foo", "V")`, no structural integrity, ...)
- No flexible dispatch on several arguments

(Some of these weaknesses disappear for code in namespaces.)

Outline

- Motivation
- S3 Classes
- S4 Classes
- Reference Classes

Basics

- Every object has exactly one class

Basics

- Every object has exactly one class
- All objects in a class must have the same structure

Basics

- Every object has exactly one class
- All objects in a class must have the same structure
- All methods for a new-style generic must have exactly the same formal arguments (could be `...`)

Basics

- Every object has exactly one class
- All objects in a class must have the same structure
- All methods for a new-style generic must have exactly the same formal arguments (could be `...`)
- Can dispatch according to the signature of arbitrarily many arguments

Outline

- Motivation
- S3 Classes
- S4 Classes
  - Classes and Inheritance
  - Generic and Method Functions
  - Group Methods
- Reference Classes

Basics

Classes are created by `setClass()`:

Basics

Classes are created by `setClass()`:

- First arg is the name of the class

Basics

Classes are created by `setClass()`:

- First arg is the name of the class
- Arg `representation` specifies the slots

```
R> setClass("fungi",
+           representation(x = "numeric", y = "numeric",
+                          species = "character"))
```

Basics (ctd.)

Creation can use existing classes:

```
R> setClass("xyloc",
+           representation(x = "numeric",
+                          y = "numeric"))
R> setClass("fungi",
+           representation("xyloc",
+                          species = "character"))
```

Basics (ctd.)

Existing classes can be examined using `getClass()`:

```
R> getClass("fungi")

Class "fungi" [in ".GlobalEnv"]

Slots:

Name:    species         x         y
Class: character   numeric   numeric

Extends: "xyloc"
```

Basics (ctd.)

New instances can be created using `new()`

```
R> f1 <- new("fungi", x = runif(5), y = runif(5),
+            species = sample(letters[1:3], 5, replace = TRUE))
```

Basics (ctd.)

Such instances can be inspected by typing their name (which calls `show()` instead of `print()`)

```
R> f1

An object of class "fungi"
Slot "species":
[1] "c" "a" "c" "b" "a"

Slot "x":
[1] 0.53827758 0.72234531 0.04228939 0.46204104 0.20706792

Slot "y":
[1] 0.69083106 0.07209983 0.51760378 0.12305639 0.45677010
```

Basics (ctd.)

The slots can be accessed using @:

```
R> f1@x
```

```
[1] 0.53827758 0.72234531 0.04228939 0.46204104 0.20706792
```

Basics (ctd.)

Typically, there would be a *creator* function:

```
R> fungi <- function(x, y, species)
+     new("fungi", x = x, y = y, species = species)
```

so that users do not need to call `new()` themselves.

Basics (ctd.)

Using the creator:

```
R> f2 <- fungi(runif(3), runif(3), letters[1 : 3])
R> f2

An object of class "fungi"
Slot "species":
[1] "a" "b" "c"

Slot "x":
[1] 0.3760097 0.0144067 0.6897521

Slot "y":
[1] 0.6598063 0.6988959 0.7950575
```

Validity checking

Can specify restrictions for “valid” objects via the `validity` argument to `setClass` or via `setValidity` (note: sets to the value of the current version.)

```
R> ## A valid "xyloc" object has the same number
R> ## of x and y values:
R> .valid_xyloc_object <- function(object) {
+     nx <- length(object@x)
+     ny <- length(object@y)
+     if(nx == ny)
+         TRUE
+     else
+         sprintf("Unequal x, y lengths: %d, %d", nx, ny)
+ }
```

Validity checking

```
R> setValidity("xyloc", .valid_xyloc_object)

Class "xyloc" [in ".GlobalEnv"]

Slots:

Name:        x       y
Class: numeric numeric

Known Subclasses: "fungi"

R> xy1 <- new("xyloc", x = runif(5), y = runif(5))
R> xy2 <- tryCatch(new("xyloc", x = runif(5), y = runif(3)),
+                  error = identity)
R> strwrap(conditionMessage(xy2))

[1] "invalid class \"xyloc\" object: Unequal x, y lengths: 5, 3"
```

Validity checking

```
R> ## We can still do bad things "directly":
R> xy1@y <- 3
R> ## However, now validObject() will throw an error ...
R> tc <- tryCatch(validObject(xy1), error = identity)
R> tc

<simpleError in validObject(xy1): invalid class "xyloc" object: Unequal

R> conditionMessage(tc)

[1] "invalid class \"xyloc\" object: Unequal x, y lengths: 5, 1"
```

Prototypes

- Classes have prototypes stored with their definition

Prototypes

- Classes have prototypes stored with their definition
- Default prototype: for each slot a new object of that class

Prototypes

- Classes have prototypes stored with their definition
- Default prototype: for each slot a new object of that class
- Other prototypes might be more appropriate; can be specified via `prototype` arg to `setClass()`

Virtual classes

- Usually have neither representation nor prototype

Virtual classes

- Usually have neither representation nor prototype
- Useful by inheritance between classes

Virtual classes

- Usually have neither representation nor prototype
- Useful by inheritance between classes
- Can be created using `"VIRTUAL"` when specifying the representation

Inheritance

Class A *inherits from* (or *extends*) class B if `is(x, "B")` is true whenever `is(x, "A")` is.

Inheritance

Class A *inherits from* (or *extends*) class B if `is(x, "B")` is true whenever `is(x, "A")` is.

Can be determined using function `extends()`:

```
R> extends("fungi", "xyloc")

[1] TRUE
```

Testing and coercion

- Use general-purpose functions `is()` and `as()` for testing and coercion:

```
R> is(f1, "xyloc")
[1] TRUE
R> as(c(1, 2, 3, 4.4), "integer")
[1] 1 2 3 4
```

Testing and coercion

- Use general-purpose functions `is()` and `as()` for testing and coercion:

```
R> is(f1, "xyloc")
[1] TRUE
R> as(c(1, 2, 3, 4.4), "integer")
[1] 1 2 3 4
```

- Corresponding information obtained from the class definition, or via `setIs()` and `setAs()`

Outline

- Motivation
- S3 Classes
- S4 Classes
  - Classes and Inheritance
  - Generic and Method Functions
  - Group Methods
- Reference Classes

Method dispatch

- Explicit setting of methods for certain *signatures* (the classes of the arguments used in the dispatch)

Method dispatch

- Explicit setting of methods for certain *signatures* (the classes of the arguments used in the dispatch)
- Can use several arguments in the dispatch

Method dispatch

- Explicit setting of methods for certain *signatures* (the classes of the arguments used in the dispatch)
- Can use several arguments in the dispatch
- Can determine the method dispatched to with `selectMethod()`

Creating generics

Typically by setting a method (which automatically makes a function generic with the old definition as the default method).

Or explicitly using `setGeneric()`.

Creating methods

By using `setMethod()`:

```
R> setMethod("show", "fungi",
+           function(object) cat("I am just a fungus.\n"))
R> f1

I am just a fungus.
```

(Note: sets to the value of the current version if a named function is used.)

For more complicated signatures:

```
R> setMethod("plot", c("numeric", "factor"), .......)
```

Outline

- Motivation
- S3 Classes
- S4 Classes
  - Classes and Inheritance
  - Generic and Method Functions
  - Group Methods
- Reference Classes

Basics

- Partially different from the Green Book (e.g., currently no `getGroupMembers()`)
- Mechanism “similar” to S3 group methods
- Can construct arbitrary groups using `setGroupGeneric()`

Group methods

Math	log, sqrt, log10, cumprod, abs, acos, acosh, asin, asinh, atan, atanh, ceiling, cos, cosh, cumsum, exp, floor, gamma, lgamma, sin, sinh, tan, tanh, trunc
Math2	round signif
Ops	Arith Compare [Logic]
Arith	+ - * / ^ %% %/%
Compare	== != < <= >= >
Summary	max, min, range, prod, sum, any, all
Complex	Arg, Conj, Im, Mod, Re

Outline

- Motivation
- S3 Classes
- S4 Classes
- Reference Classes

Basics

Main differences between reference classes and S3 and S4:

- Refclass objects use message-passing OO
- Refclass objects are mutable: the usual R copy on modify semantics do not apply

Thus, the refclass object system behaves more like Java or C#.

Use them only where mutable state is absolutely required!

Basics

A reference class has three main components, given by three arguments to `setRefClass`:

- `contains`, the classes which the class inherits from.

Basics

A reference class has three main components, given by three arguments to `setRefClass`:

- `contains`, the classes which the class inherits from.
- `fields` are the equivalent of slots in S4. They can be specified as a vector of field names, or a named list of field types.

Basics

A reference class has three main components, given by three arguments to `setRefClass`:

- `contains`, the classes which the class inherits from.
- `fields` are the equivalent of slots in S4. They can be specified as a vector of field names, or a named list of field types.
- `methods` are functions that operate within the context of the object and can modify its fields.

Example

A simple reference class for bank accounts with methods for withdrawal and deposit:

```
R> setRefClass("Account",
+              fields = list(balance = "numeric"),
+              methods =
+              list(deposit =
+                   function(amount) {
+                       ## The following string documents the method
+                       "Deposit the given amount from the account"
+                       balance <<- balance + amount
+                   },
+                   withdraw =
+                   function(amount) {
+                       "Withdraw the given amount from the account"
+                       balance <<- balance - amount
+                   }))
```

Example

What is returned by `setRefClass`, or can be retrieved by `getRefClass`, is a *reference class generator*:

```
R> (Account <- getRefClass("Account"))

Generator for class "Account":

Class fields:

Name:  balance
Class: numeric

Class Methods:
     "withdraw", "deposit", "field", "trace", "getRefClass",
     "initFields", "copy", "callSuper", ".objectPackage", "export",
     "untrace", "getClass", "show", "usingMethods", ".objectParent",
     "import"

Reference Superclasses:
     "envRefClass"
```

Example

The reference class generator has methods:

- `new()` for creating new objects of that class. Takes named arguments specifying initial values for the fields.

Example

The reference class generator has methods:

- `new()` for creating new objects of that class. Takes named arguments specifying initial values for the fields.
- `methods()` for modifying existing or adding new methods

Example

The reference class generator has methods:

- `new()` for creating new objects of that class. Takes named arguments specifying initial values for the fields.
- `methods()` for modifying existing or adding new methods
- `help()` for getting help about methods

Example

The reference class generator has methods:

- `new()` for creating new objects of that class. Takes named arguments specifying initial values for the fields.
- `methods()` for modifying existing or adding new methods
- `help()` for getting help about methods
- `fields()` to get a list of fields defined for class

Example

The reference class generator has methods:

- `new()` for creating new objects of that class. Takes named arguments specifying initial values for the fields.
- `methods()` for modifying existing or adding new methods
- `help()` for getting help about methods
- `fields()` to get a list of fields defined for class
- `lock()` locks the named fields so that their value can only be set once

Example

We can use the generator to create an account object:

```
R> a1 <- Account$new(balance = 100)
R> a1

Reference class object of class "Account"
Field "balance":
[1] 100
```

We can ask the generator for help:

```
R> Account$help("withdraw")

Call:
$withdraw(amount)


Withdraw the given amount from the account
```

Example

We can then make deposits and withdrawals:

```
R> a1$withdraw(50)
R> a1

Reference class object of class "Account"
Field "balance":
[1] 50
```

To access the balance:

```
R> a1$balance

[1] 50

R> a1$field("balance")

[1] 50
```

Example

The most important property of refclass objects is that they are mutable, or equivalently they have reference semantics:

```
R> a2 <- a1
R> a2$deposit(10)
R> a1

Reference class object of class "Account"
Field "balance":
[1] 60
```

(In the example, copies will always refer to the same account.)

Cool stuff

```
R> setRefClass("Dist")
R> setRefClass("DistUniform",
+             c("a", "b"),
+             "Dist",
+             methods =
+             list(mean =
+                  function() {
+                      (a + b) / 2
+                  }))
R> DistUniform <- getRefClass("DistUniform")
R> U01 <- DistUniform$new(a = 0, b = 1)
R> U01$mean()

[1] 0.5
```

(Could add other sub-classes for observed or simulated values.)