
# Object-Oriented Programming

Kurt Hornik

# Overview

- Motivation
- S3 Classes
- S4 Classes

# Motivation

## 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, . . . )


# S3 Classes

# 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`


# Classes

# Manipulating the class info

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

```
class(x) <- "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)


# Generic and Method Functions

# 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.,

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

# Creating methods

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

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

# Explicitly invoking inheritance

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

```
test <- function(x) UseMethod("test")
test.c1 <- function(x) { cat("c1\n"); NextMethod(); x }
test.c1 <- function(x) { cat("c2\n"); NextMethod(); x }
test.c1 <- function(x) { cat("c3\n"); NextMethod(); x }
x <- 1; class(x) <- c("c1", "c2"); 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.foo` and `as.foo`. E.g.,

```
is.foo <- function(x) inherits(x, "foo")
as.foo <- function(x) if(is.foo(x)) x else foo(x)
```

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

# Group Methods

# Motivation

- 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

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


# Odds and ends

# 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
- 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 (“multiple inheritance”)

# S4 Classes

# 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 ...)
- Multiple inheritance


# Creating Classes

# Basics

- Classes are created by `setClass`
- First arg is the name of the class
- Arg `representation` specifies the slots

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

# Basics (ctd.)

- Creation can use existing classes:

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

- Existing classes can be examined using `getClass`

```
getClass("fungi")
```

# Basics (ctd.)

- New instances can be created using `new`

```
f1 <- new("fungi", x = runif(20), y = runif(20),
          species = sample(letters[1:5], 20,
                           replace = TRUE))
```

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

```
f1
```

# Basics (ctd.)

- Typically, there would be a *creator* function:

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

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

# Validity checking

Can specify restrictions for "valid" objects via

```
setClass("xyloc",
         representation(x = "numeric", y = "numeric"),
         validity = .validFungi)
setValidity("fungi", .validFungi)
```

(Note: sets to the value of the current version.)

# 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
- Useful by inheritance between classes
- Can be created using `"VIRTUAL"` when specifying the representation


# Inheritance

# 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`:

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

# Testing and coercion

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

```
is(f1, "xyloc")
as(c(1, 2, 3, 4.4), "integer")
```

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


# Generic and Method Functions

# 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`:

```
setMethod("show", "fungi",
          function(object) cat("I am just a fungus.\n"))
setMethod("plot", c("numeric", "factor"), .......)
```

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

# Group Methods

# 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 |