MONASH University

MONASH BUSINESS SCHOOL

ETC4500/ETC5450
Advanced R programming

Week 8: Object-oriented Programming

Outline

Outline

Outline

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Programming paradigms

Functional programming (W5)

- Functions are created and used like any other object.
- Output should only depend on the function's inputs.

Literate programming (W6)

- Natural language is interspersed with code.
- Aimed at prioritising documentation/comments.
- Now used to create reproducible reports/documents.

Programming paradigms

Reactive programming (W7)

- Objects are expressed using code based on inputs.
- When inputs change, the object's value updates.

Programming paradigms

Reactive programming (W7)

- Objects are expressed using code based on inputs.
- When inputs change, the object's value updates.

Object-oriented programming (W8-W9)

- Functions are associated with object types.
- Methods of the same 'function' produce object-specific output.

Outline

Object oriented programming

- **Encapsulation**: Bundles data and methods in a class, restricting access to internal details.
- **Abstraction**: Simplifies complexity by exposing only essential features of an object.
- **Polymorphism**: Allows the same function to operate differently on different object types.
- **Inheritance**: Enables a new class to inherit properties and behaviors from an existing class.

Object oriented programming

Inheritance is primarily useful for structuring data infrastructure by allowing reuse and extension of existing classes.

Encapsulation helps protect object integrity by restricting access to internal states.

Polymorphism enables flexibility by allowing a single interface to operate on various data types.

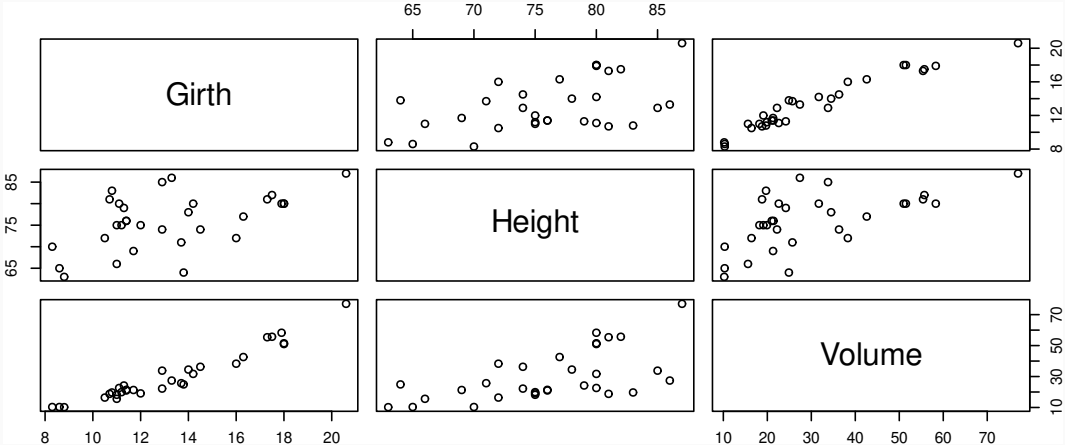
Abstraction simplifies complexity by highlighting essential features, making systems easier to understand and use.

Generic functions and methods

A simple example: `plot`

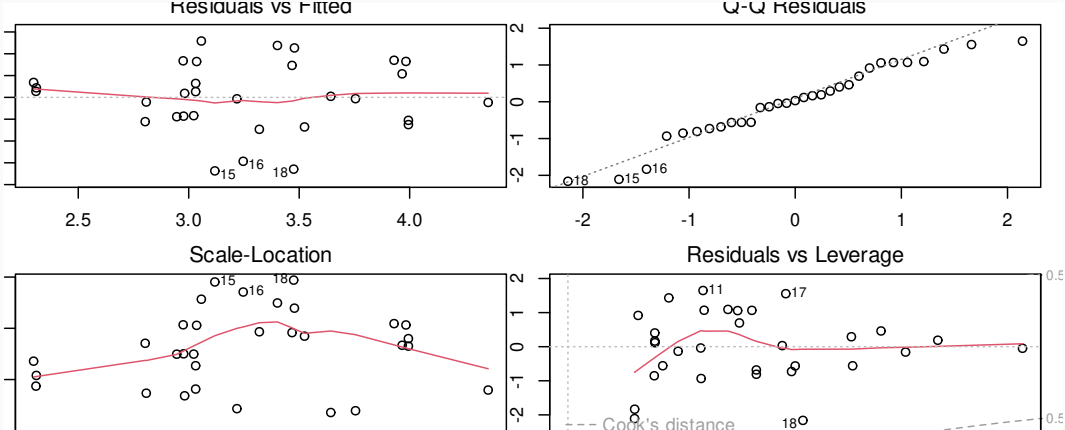
Generic functions and methods

```
plot(trees)
```



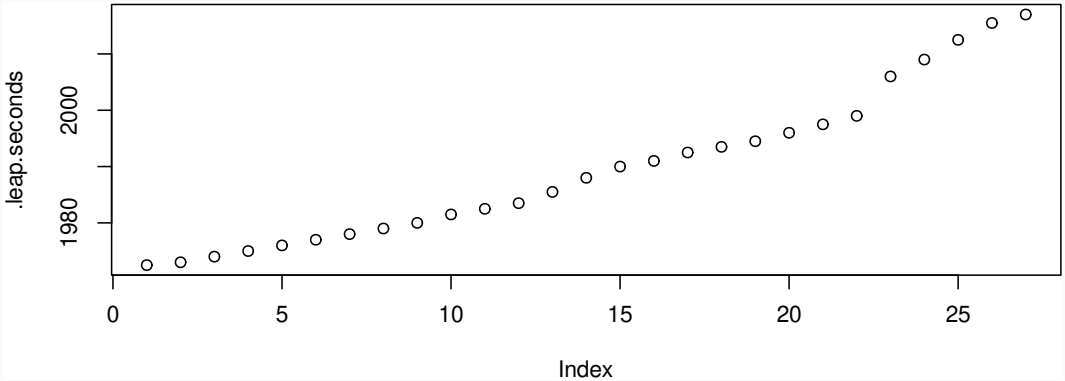
Generic functions and methods

```
m<-lm(log(Volume)~log(Girth)+log(Height),
      data=trees)
par(mfrow=c(2,2),mar=c(3,1,1,1))
plot(m)
```



Generic functions and methods

```
plot(.leap.seconds)
```



Generic functions and methods

```
plot(nhtemp)
```

Generic functions and methods

How does `plot()` work?

- Giant `switch` statement...
- Lots of `if` statements...
- How does the behaviour update when you load packages?
- ???

Generic functions and methods

How does `plot()` work?

- Giant `switch` statement...
- Lots of `if` statements...
- How does the behaviour update when you load packages?
- ???

S3 generic functions and methods!

Object systems

R has **a lot** of object systems

- S3
- [S3 `vctrs`]
- S4
- R6
- R.oo, proto, ggproto,
- R7
- S7

Object oriented programming

S3

- The OO system used by most of CRAN.
- Very simple (and 'limited') compared to other systems.

Object oriented programming

S3

- The OO system used by most of CRAN.
- Very simple (and ‘limited’) compared to other systems.

vctrs

- Builds upon S3 to make creating vectors easier.
- Good practices inherited by default.

Object oriented programming

S4

- Formal class definitions with validation.
- Supports multiple inheritance and method dispatch.

Object oriented programming

S4

- Formal class definitions with validation.
- Supports multiple inheritance and method dispatch.

S7

- Planned to be the successor of S3 and S4.
- More general than S3, but still easy to use.

Object oriented programming

R6

- Provides reference semantics for mutable objects.
- Simple and efficient compared to reference classes.

Object oriented programming

R6

- Provides reference semantics for mutable objects.
- Simple and efficient compared to reference classes.

ggproto

- Used in ggplot2 for extensibility.
- Supports inheritance and method overloading.

Outline

S3

Main topic for today

- easy to start writing
- no safeguards
- especially good for simple, small-medium projects
- can be used for large projects with a lot of attention to documentation and communication
- limited use of inheritance
- basis of tidyverse and most of CRAN

S3 with `vctrs`

A helpful package for making different sorts of S3 vectors

- handles a lot of formatting and subsetting details
- allows for binary operators
- useful if you want your vectors in a tibble
- enforces some safeguards

We'll learn more about `vctrs` next week!

Back to the `plot` function…

- `plot()` doesn't *do* anything
- All the work is done by *methods* for different *classes*
- Methods are just ordinary functions
- When you call `plot`, R calls the appropriate `plot` method

S3: Generic functions

- Generic functions don't *do* anything
- All the work is done by *methods* for different types of object
- Methods are just ordinary functions
  - with declarations in a package NAMESPACE
  - or R can guess based on function name

When you call the generic function R calls the appropriate method

S3: Generics and methods

Your turn!

Investigate these functions.

```
print
methods("print")
stats:::print.acf
tools:::print.CRAN_package_reverse_dependencies_and_views
plot
methods("plot")
plot.ts
stats:::plot.lm
```

How do generic functions relate to methods?

Also, try `methods("plot")` after loading another package.

S3: Classes

- S3 classes are attributes that specify which method to use
- The `class()` function can access (and modify) an object's class
- Classed S3 objects are typically produced with `structure()`

For example,

```
x <- structure(83, class = "grade")
class(x)
```

```
[1] "grade"
```

```
x
```

```
[1] 83
```

S3: Methods

- *methods* that actually do the work 'belong to' *generic functions*
- This is unusual: most other OOP systems (Java, C++, Python) have methods belonging to data objects
- Important in R because functions are first-class objects (Week 5)
- Useful for functional programming with objects

S3: Creating a method

```
print.grade <- function(x, ...){
  letter <- if (x < 50) "N"
  else if (x < 60) "P"
  else if (x < 70) "C"
  else if (x < 80) "D"
  else "HD"
  cat(x," [", letter, "]", sep = "")
  invisible(x)
}
x
```

```
83 [HD]
```

Creating an S3 generic

S3 generics are work like any ordinary function, but they include `UseMethod()` which calls the appropriate method.

Your turn!

Create an S3 generic called “reverse”.
This function will reverse objects. For example,

- `reverse("stressed")` becomes `"desserts"`,
- `reverse(7919)` becomes `9197`,
- `reverse(1.9599)` becomes `9959.1`.

Writing S3 methods

An S3 method is an ordinary function with some constraints:

- The function's name is of the form `<generic>.<class>`,
- The function's arguments match the generic's arguments,
- The function is registered as an S3 method (for packages).

This looks like:

```
#' Documentation for the method
#' @method <generic> <class>
<generic>.<class> <- function(<generic args>, <method args>, ...) {
  # The code for the method
}
```

Writing S3 methods

Your turn!

Write methods for reversing `character`, `integer`, and `double` objects.

- `reverse("stressed")` becomes `"desserts"`,
- `reverse(7919L)` becomes `9197L`,
- `reverse(1.9599)` becomes `9959.1`.

Hint: `stringi::stri_reverse()` will reverse a string.
The `integer` and `double` methods should return an integer and double respectively.

S3: `.default` methods

Default methods are called when there is no specific method for the object (no class, or no matching class).

Some examples include:

- `mean.default`
- `summary.default`
- `head.default`

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

Your turn!

Question: what should the default behaviour be?

Writing S3 defaults

What if we tried to reverse the current date; `reverse(Sys.Date())`?

Your turn!

Question: what should the default behaviour be?

- Raise an error?
- Return a reversed string?
- Something else entirely?

S3: Defining classes

The S3 class system is simple!

- R doesn't care what `class` you attach to an object
- **You** have to care
- `class(x) <- "lm"` makes R call `lm` methods on `x`
- **You** are responsible for these methods being appropriate
- Documentation is important
- No real enforcement of encapsulation

S3: Classed objects

You can class any object, including:

- vectors plus attributes (`ts`, `POSIXct`, matrix)
- lists plus attributes (`lm`, `data.frame`)
- environments plus attributes

Your turn!

Use `unclass()` and `str()` to explore classed objects, e.g.:

```
unclass(.leap.seconds)
unclass(nhtemp)
unclass(trees)
m<-lm(log(Volume)~log(Girth)+log(Height),data=trees)
str(m)
```

S3: Constructors functions

These functions return classed S3 objects. They should handle input validation and be user-friendly.

Constructor functions typically come in two forms:

- complex: `tibble`, `lm`, `acf`, `svydesign`
- pure: `new_factor`, `new_difftime`

Pure constructor functions simply validate inputs and produce the classed object, while complex constructor functions involve calculations.

Creating your own S3 objects

The `structure()` function is usually used within packages.

- `lm()` returns a list with class `"lm"`, and
- `tibble()` returns a list classed `"tbl_df"`, `"tbl"`, and `"data.frame"`.

Creating your own S3 objects

The `structure()` function is usually used within packages.

- `lm()` returns a list with class `"lm"`, and
- `tibble()` returns a list classed `"tbl_df"`, `"tbl"`, and `"data.frame"`.

Your turn!

Create `fraction()`, which returns `fraction` objects.
The underlying data type is a list containing two vectors for the two arguments: `numerator` and `denominator`.
This function should check that the inputs are suitable.

Creating your own S3 objects

```r
fraction <- function(numerator, denominator) {
  if (!is.numeric(numerator) || !is.numeric(denominator)) {
    stop("Both numerator and denominator must be numeric.")
  }
  if (denominator == 0) {
    stop("Denominator cannot be zero.")
  }

  structure(
    list(numerator = numerator, denominator = denominator),
    class = "fraction"
  )
}
```

Creating your own S3 objects

The `fraction` class doesn't yet have any methods, so it inherits methods from its list type.

```
e <- fraction(numerator = 2721, denominator = 1001)
print(e)
```

```
$numerator
[1] 2721

$denominator
[1] 1001

attr(,"class")
[1] "fraction"
```

Creating your own S3 objects

Usually we would create a method for printing S3 objects.

```
print.fraction <- function(x, ...) {
  paste(x$numerator, x$denominator, sep = "/")
}
print(e)
```

```
[1] "2721/1001"
```

Creating your own S3 objects

Your turn!

Create a `reverse()` method for the `fraction` object class, which inverts the numerator and denominator.

Finished early?
Write a method for converting a `fraction` into a number.

S3: Method dispatch

Method dispatch describes the process of calling the appropriate method for the object's class.

This mostly matches `class()`, but not always for some primitive R object types. `sloop::s3_class()` shows the extra s3 dispatch classes.

```
> s3_class(1)
[1] "double"  "numeric"
> s3_class(matrix(1,1,1))
[1] "matrix"  "double"  "numeric"
> class(1)
[1] "numeric"
> class(matrix(1,1,1))
[1] "matrix" "array"
```

S3: Naming ambiguity

- `t` is a generic
- `t.test` is a generic
- `t.test.formula` is a method for `t.test`
- `t.data.frame` is a method for `t`
- `list` is not generic
- `list.files` isn't a method

Avoid using `.` as a word separator in function names that aren't methods.

Use `camelCase` or `snake_case` or some other consistent approach

S3: Inheritance

The `class` attribute of an object can have multiple elements

- `UseMethod()` uses the first method that matches, or `default`
- `NextMethod()` uses the next method that matches

S3: Polite conduct

- if you define a new generic, you can define methods for new and existing classes
- if you define a new class, you can define methods for new and existing generics
- don't define methods for someone else's class and generic (ask them)
- try not to define a generic with the same name as an existing one

Outline

S4

S4 requires classes and methods to be registered in R code (not just in packages)

- `setClass` defines the structure of a class
- `new` creates a new object from a class
- `setMethod` defines a method

It's possible to ask an object what methods it supports and get a reliable response.

S4 also allows multiple inheritance and multiple dispatch

S4: Bioconductor

- Package system for high-throughput molecular biology
- Large data
- Structured data
- Annotated data
- New data types/structures all the time

It needs consistent infrastructure and large-scale collaboration: S4

bioconductor.org

S4: Multiple dispatch

Choosing a method based on the class of more than one argument

- not very often useful
- important for matrices
- can be useful for plots

S4: Multiple inheritance

`AnnDbObjBimap` is a class for storing look-up tables between different genomic identifiers (eg from different manufacturers)

It is

- (by purpose) a two-way lookup object (`BiMap`)
- (by construction) an object containing a SQLite database (`DbObj`)

so it inherits generic functions from both these parents

S4: Creating a class

The structure of your S4 class is defined with `setClass()`.

```
setClass(
  "StudentGrades",
  slots = list(
    name = "character",
    grades = "numeric"
  )
)
```

S4: Creating S4 objects

S4 objects are created with the `new()` function.

```r
studentGrades <- function(name, grades) {
  if (!is.character(name) || length(name) != 1) {
    stop("Name must be a single string.")
  }
  if (!is.numeric(grades)) {
    stop("Grades must be numeric.")
  }

  new("StudentGrades", name = name, grades = grades)
}
```

S4: Creating methods

Methods are registered to S4 classes with `setGeneric()` and `setMethod()`.

```
setGeneric("averageGrade", function(object) {
  standardGeneric("averageGrade")
})
```

```
[1] "averageGrade"
```

```
setMethod("averageGrade", "StudentGrades", function(object) {
  mean(object@grades)
})
```

S4: Using S4 objects

```
student <- studentGrades("Alice", c(85, 90, 78))
print(student)
```

```
An object of class "StudentGrades"
Slot "name":
[1] "Alice"

Slot "grades":
[1] 85 90 78
```

```
average <- averageGrade(student)
print(paste("Average Grade:", average))
```

```
[1] "Average Grade: 84.3333333333333"
```

S4: Accessing S4 slots

Contents of an S4 object are extracted with `@`.

For example, the student's name can be obtained with:

```
student@name
```

```
[1] "Alice"
```

Outline

S7

S7 = S3 + S4

It aims to maintain the simplicity of S3, while adding useful features from S4.

(and unify CRAN and Bioconductor packages!)

It's not yet in R-Core, but it can be used via the `S7` package.

```
library(S7)
```

S7: Creating a class

Like S4, S7 starts by defining the data structure.

```
student <- new_class(
  name = "student",
  properties = list(
    name = class_character,
    grades = class_double
  )
)
```

S7: Self-validation

S7 additionally supports property validation

```
student <- new_class(
  name = "student",
  properties = list(
    name = class_character,
    grades = class_double
  ),
  validator = function(self) {
    if (any(self@grades < 0 | self@grades > 100)) {
      "@grades must be between 0 and 100"
    }
  }
)
```

S7: S7 classes are also constructors

The S7 class `student` is also a (pure) constructor function.

```
x <- student(name = "Alice", grades = c(85, 90, 78))
x
```

```
<student>
 @ name  : chr "Alice"
 @ grades: num [1:3] 85 90 78
```

The `validator` prevents invalid grades.

```
student(name = "Mitch", grades = c(-10, 140))
```

```
Error: <student> object is invalid:
- @grades must be between 0 and 100
```

S7: Creating generics

S7 generics are created with `new_generic()`.

```
best_grade <- new_generic("best_grade", dispatch_args = "x")
```

Here we explicitly specify which argument(s) are used in finding the appropriate method. Double (or multiple) dispatch is supported!

S7: Creating methods

S7 methods are created with `method<-`:

```
method(best_grade, student) <- function(x) {
  max(x@grades)
}
best_grade(x)
```

```
[1] 90
```