9. Introduction to the S4 Object System

CT5102 - J. Duggan

S4 (Wickam 2019, Chapter 15)

- A more formal approach to functional OOP
- Underlying ideas similar to S3, but implementation is much stricter
- Makes use of specialised functions for:
  - Creating classes **setClass()**
  - Generics **setGeneric()**
  - Methods **setMethod()**
- Provides multiple inheritance and multiple dispatch
- A new component of S4 is the slot, a named component of an object that can be accessed using @ (pronounced at)
- Include **library(methods)** when using S4

Basics

```
setClass("Person",
         slots = c(
           name = "character",
           age  = "numeric"
         )
)

john <- new("Person", name = "John Smith", age=35)
```

Given an S4 object, you can see its class with **is()** and access its slots with **@** and **slot()**

Calling S4 functions

```
is(john)
```

```
## [1] "Person"
```

```
john@name
```

```
## [1] "John Smith"
```

```
john@age
```

```
## [1] 35
```

```
slot(john,"age")
```

```
## [1] 35
```

```
slot(john,"name")
```

```
## [1] "John Smith"
```

Accessing slots: Guidelines

- Generally, onlyuse @ in your methods
- Look for accessor functions that allow you to safely set and get slot values
- When you develop a class, provide your own access functions
- Creating a setter and getter for the age slot by
  - Creating getter and setter generics usiing **setGeneric()**
  - Defining methods with **setMethod()**

getter for slot age

```
setGeneric("age", function(x)standardGeneric("age")) # getter
```

```
## [1] "age"
```

```
setMethod("age", "Person", function(x) x@age)

age(john)
```

```
## [1] 35
```

setter for slot age

```
setGeneric("age<-", function(x, value)standardGeneric("age<-")
```

```
## [1] "age<-"
```

```
setMethod("age<-", "Person", function(x, value){
  x@age <- value
  x
})
age(john) <- 29
age(john)
```

```
## [1] 29
```

Creating S4 Classes

- To define an S4 class, call setClass() with three arguments
  - The class **name**. By convention S4 class names use UpperCamelCase
  - A named character vector that describes the names and classes of the **slots** (fields). The pseudo-class ANY allows a slot to accept objects of any type
  - A **prototype**, a list of default values for each type (optional but should be provided)

S4 class with 3 arguments

```
setClass("Person",
         slots = c(
           name = "character",
           age  = "numeric"
         ),
         prototype=list(
           name = NA_character_,
           age  = NA_real_
         )
)

test <- new("Person", name = "A.N. Other")
str(test)
```

```
## Formal class 'Person' [package ".GlobalEnv"] with 2 slots
##   ..@ name: chr "A.N. Other"
##   ..@ age : num NA
```

S4 class with inheritance

```
setClass("Employee",
         contains = "Person",
         slots = c(
           boss = "Person"
         ),
         prototype=list(
           boss = new("Person")
         )
)
str(new("Employee"))
```

```
## Formal class 'Employee' [package ".GlobalEnv"] with 3 slots
##   ..@ boss:Formal class 'Person' [package ".GlobalEnv"] wit
##   .. .. ..@ name: chr NA
##   .. .. ..@ age : num NA
##   ..@ name: chr NA
```

Helper

User facing classes should always be paired with a user-friendly helper. A helper should always:

- Have the same name as the class
- Have a thoughtfully crafted user interface with carefully chosen default values
- Create error messages tailored towards an end user
- Finish by calling **methods::new()**

Example

```
Person <- function(name, age=NA){
  age <- as.double(age)
  new("Person", name=name, age=age)
}

p1 <- Person("A.N. Other")
p1
```

```
## An object of class "Person"
## Slot "name":
## [1] "A.N. Other"
##
## Slot "age":
## [1] NA
```

Validator

- The constructor automatically checks that the slots have correct classes
- More complicated checks may be required
- For example, name and age may have to be the same vector length
- We can write a validator with **setValidity()**

```
setValidity("Person", function(object) {
  if(length(object@name) != length(object@age)){
    "@name and @age must be the same length"
  }else{
    TRUE
  }
})
```

Generics and Methods

- The job of a generic is to perform method dispatch (find the specific implementation for the combination of classes passed to the generic)
- To create a new S4 generic, call **setGeneric()** with a function that calls **stanardGeneric()**
- By convention, new S4 generics should use lowerCamelCase.
- It is bad practice to use {} as it triggers a special case that is more computationally expensive

```
setGeneric("myGeneric",
           function(x) standardGeneric("myGeneric"))
```

Generics and Methods

- A generic isn't useful without some methods, and in S4 methods are defined with **setMethod()**
- There are three important arguments:
  - The name of the generic
  - The name of the class
  - The method itself

```
setMethod("myGeneric", "Person", function(x){
  # method implementation
})
```

Show method

- The most commonly defined S4 method that controls printing is **show()**.
- To define a method for an existing generic, you must first determine its arguments
- Our show method needs to have a single argument object

```
args(getGeneric("show"))
```

```
## function (object)
## NULL
```

Writing the method

```
setMethod("show", "Person", function (object){
  cat(is(object)[[1]], "\n",
  "    Name: ", object@name, "\n",
  "    Age:  ", object@age, "\n",
  sep="")
})

p1
```

```
## Person
##     Name: A.N. Other
##     Age:  NA
```

Accessors - getter function

- Slots should be considered an internal implementation detail: user code should avoid accessing them directly
- Typically, a generic will be defined so that multiple classes can use the same interface

```
setGeneric("name", function(x) standardGeneric("name"))
```

```
## [1] "name"
```

```
setMethod("name", "Person", function(x) x@name)

name(p1)
```

```
## [1] "A.N. Other"
```

Accessors - setter function

- If the slot is writeable, a setter function should be provided
- Should also include **validObject()** in the setter

```
setGeneric("name<-", function(x, value)
             standardGeneric("name<-"))
```

```
## [1] "name<-"
```

```
setMethod("name<-", "Person", function(x, value){
  x@name <- value
  x})

name(p1) <- "Test Name"
name(p1)
```

```
## [1] "Test Name"
```

Exercises and Further features of S4

- Exercises
  - Add age() accessors for the Person class
  - In ther definition of the generic, why is it necessary to repeat the name of the generic twice
- Method Dispatch
  - Multiple inheritance - a class can have multiple parents
  - Multiple dispatch - a generic can use multiple arguments to pick a method

S4 Summary