R Object Oriented Programming

Robert Stojnić rs550@cam.ac.uk
Laurent Gatto lg390@cam.ac.uk

Cambridge System Biology Centre
University of Cambridge

24 - 25 January 2013

General overview

- R object-oriented programming – in details; after this, you should be able to tackle more complicated OO designs.

Prerequisites

- good knowledge of R (data types, functions, scripting ...)
- object-oriented programming knowledge helpful but not essential

Plan

Plan

Course agenda

- Object-oriented programming in R: S3 and S4 class systems
- Package development in R: creating and documenting packages
- Other advanced topics: testing, debugging, profiling, C/C++ interface, parallel computation

Objectives

By the end of the course you should have created a working package written in the S4 class system.

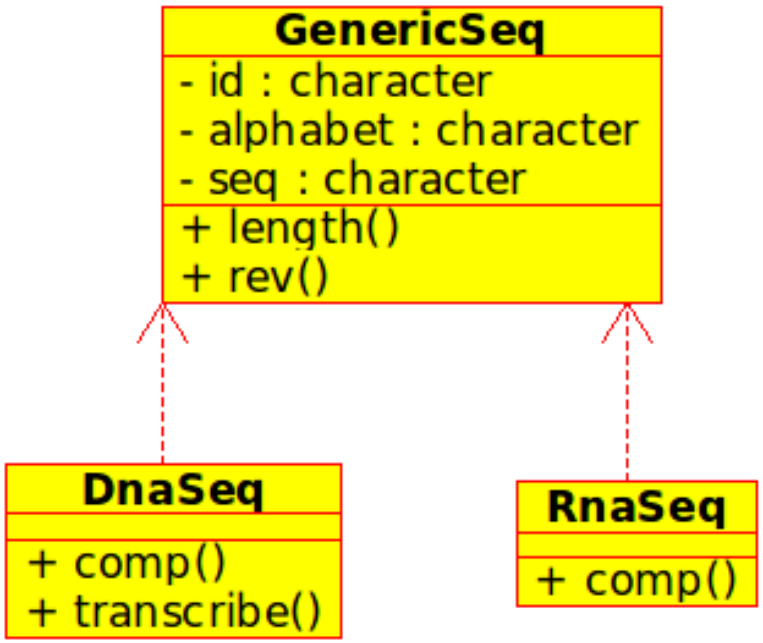
You should be able to use the code as a template for your own work. Our example has been chosen for demonstrative purposes.

Course working example: "sequences" package

Working example

We will make a simple package to handle sequence data.

This package will be able to load a FASTA file and based on sequence type do some operations, like finding the sequence length or reverse sequence. For simplicity we will manipulate single sequences only.



UML class diagram for the "sequences" package

Plan

Basic R functions (size reflects frequency of usage)

Defining functions in R

Simple function with 4 arguments:

```
> # Function to calculate area of rectangle
> area <- function(x1, y1, x2, y2) {
+     abs(x2 - x1) * abs(y2 - y1)
+ }
> area(0, 0, 5, 5)

[1] 25
```

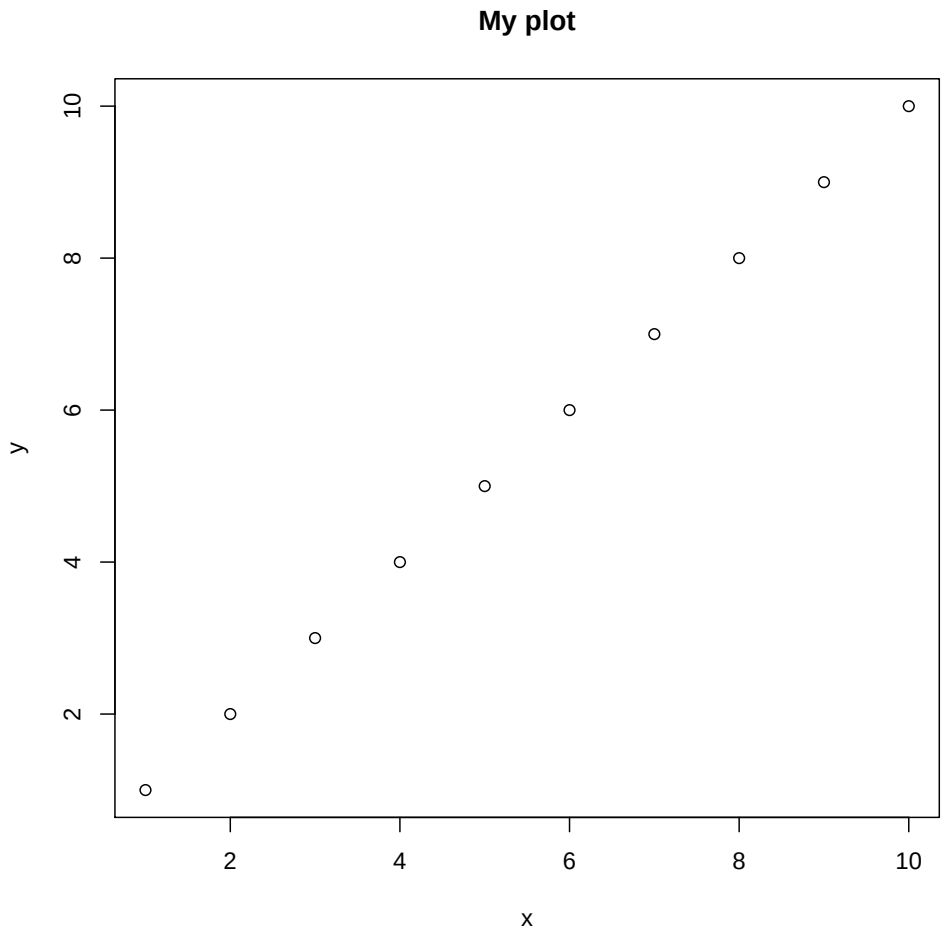
Special argument "..." for any:

```
> # Plot with a message before the plot
> plotMsg <- function(x, y, ...) {
+     cat("Plotting", length(x), "data points!\n")
+     plot(x, y, ...)
+ }
```

Output of plotMsg()

```
> plotMsg(1:10, 1:10, main="My plot")

Plotting 10 data points!
```



Useful R function 1/2

- readLines() - reads raw lines of text from a file
- nchar() - gives number of characters in a string

```
> nchar("Some text")

[1] 9
```

- strsplit() - split a string by some separator

```
> strsplit("Some text", " ")

[[1]]
[1] "Some" "text"

> strsplit("Some text", "")

[[1]]
[1] "S" "o" "m" "e" " " "t" "e" "x" "t"
```

- unique() - unique elements of a vector

```
> unique(c(1, 1, 2, 2, 3))

[1] 1 2 3

> unique(c("a", "b", "a"))

[1] "a" "b"
```

Useful R function 2/2

- `grep()` - find which elements of vector match regular expression

```
> grep("[AT]+", c("CGC", "TAT", "TATCATA"))

[1] 2 3
```

- `sub()` - replace matches to regular expression

```
> sub("[AT]+", "-", c("CGC", "TAT", "TATCATA"))

[1] "CGC"   "-"     "-CATA"
```

- `chartr()` - translate a string by replacing individual characters

```
> chartr("TA", "AT", "TATCTA")

[1] "ATACAT"
```

- `rev()` - reverse ordering in a vector

```
> rev(c("TAT", "ATT", "TTT"))

[1] "TTT" "ATT" "TAT"
```

- `paste()` - concatenate variables into a string representation

```
> paste(c("A", "T", "A"), collapse = "")

[1] "ATA"
```

Lists in R

List is a data structure that can hold a vector of any other variables.

```
> x <- list(a = 10, b = "text")
> x$a

[1] 10

> x[["b"]]

[1] "text"

> x[[1]]

[1] 10

> names(x)

[1] "a" "b"
```

Everything in R has a class

Everything in R has a type - in object oriented programming called a **class**.

```
> class(c(1, 2, 3))

[1] "numeric"

> class("Some text")

[1] "character"

> class(matrix(0, nrow = 10, ncol = 10))

[1] "matrix"

> class(plot)

[1] "function"

> class(table(1:4, 1:4))

[1] "table"
```

Recommended coding standards

Coding standards

- Use <- for assignment rather than =.
- Avoid long lines (80 characters).
- Use spaces for identation (2 or 4).
- No semi-colomns (unless you have several expression in a line).
- Start names with upper case for classes, lower for the rest.
- Use syntax highlighting

Plan

Object-oriented Programming (OOP)

Object-oriented vs Procedural programming

- OOP introduced in 1970s in Smalltalk but gained wider popularity in 1990s with programming languages like C++ and Delphi

- Traditional (procedural) programming - data and functions decoupled
- Object-oriented programming - data and functions tied together in **objects**

OOP concepts

- **Abstraction** - related data is stored and handled together
- **Inheritance** - code reuse by hierarchy of more-to-less general object types (classes)
- **Polymorphism** - the most appropriate function is called based on the dataset (e.g various `plot` functions)

Procedural vs Object-oriented Programming

Procedural programming

```
> area <-
+   function(x1,y1,x2,y2){
+     abs(x2-x1)*abs(y2-y1)
+   }
> area(0, 0, 5, 5)

[1] 25
```

Object-oriented programming

```
> setClass("Rectangle",
+          representation = representation(
+            x1 = "numeric",
+            y1 = "numeric",
+            x2 = "numeric",
+            y2 = "numeric")
+          )
> setGeneric("area",
+            function(obj) standardGeneric("area"))

[1] "area"

> setMethod("area", "Rectangle", function(obj) {
+   abs(obj@x2 - obj@x1) * abs(obj@y2 - obj@y1)
+ })

[1] "area"

> rect = new("Rectangle", x1=0, y1=0, x2=5, y2=5)
> area(rect)
```

OOP in R (1)

S3

Older and less formal framework with no explicit class definitions. Many parts of base Ruse S3, e.g. plotting, linear modelling, ...

- limited introspection, single inheritance, single dispatch, instance-based

S4

Full-fledged object-oriented framework, de-facto standard for most modern packages and required for Bioconductor packages.

- introspection, multiple inheritance, multiple dispatch (introduces a small overhead)

S4 Reference classes

Introduced in R-2.12

- mutable objects, single inheritance, single dispatch, fields and methods in class definition, methods associated with classes (rather than generics)

Working example revisited

Working example for this course will be **manipulating DNA/RNA sequence data**.

Functions we would like to have:

- `readFasta()` - read in a single sequence from a FASTA file
- `id()`, `seq()` - return the ID of sequence and the sequence (accessors)
- `rev()` - return reverse DNA/RNA sequence
- `length()` - return DNA/RNA sequence length
- `comp()` - return complementary DNA/RNA sequence
- `transcribe()` - return RNA sequence for DNA sequence

Goal

The final product should be an R package using S4 framework. But we need to get there, so lets start with a procedural and S3 implementation...

Plan

`readFasta()` input file

We will start with the implementation of `readFasta()`. This function should load the data from a FASTA file and somehow represent it in `R`.

A sample FASTA file:

```
> example dna sequence
agcatacgacgactacgacactacgacatcagacactacagactactac
gactacagacatcagacactacatatttacatcatcagagattatatta
acatcagacatcgacacatcatcatcagcatcat
```

Sequence description

Notice that a sequence is described by the:

- name (`example dna sequence`)
- nucleotide sequence
- sequence alphabet (in case of DNA ATGC, for RNA AUGC)

readFasta() implementation

Read in a sequence from FASTA file and return the `id`, `sequence` and `alphabet` in a list:

```
> readFasta <- function(infile) {
+     lines <- readLines(infile)
+     header <- grep("^>", lines)
+     if (length(header) > 1) {
+         warning("Reading first sequence only.")
+         lines <- lines[header[1]:(header[2] - 1)]
+         header <- header[1]
+     }
+     id <- sub("^> *", "", lines[header], perl = TRUE)
+     sequence <- toupper(paste(lines[(header + 1):length(lines)],
+         collapse = ""))
+     alphabet <- unique(strsplit(sequence, "")[[1]])
+     return.value <- list(id = id, sequence = sequence,
+         alphabet = alphabet)
+     class(return.value) <- "GenericSeq"
+     return.value
+ }
```

S3 objects

```
> s <- readFasta("aDnaSeq.fasta")
> s

$id
[1] "example dna sequence"

$sequence
[1] "AGCATACGACGACTACGACACTACGACATCAGACACTACAGACTACTACGACTACAGACATCAGACACTACATATTTA

$alphabet
[1] "A" "G" "C" "T"

attr(,"class")
[1] "GenericSeq"

> names(s)

[1] "id"       "sequence" "alphabet"
```

S3 object definition

- Any variable that has a "`class`" attribute is an S3 object.
- Now we can write class-specific functions - **methods**.

S3 generics and dispatch

```
> x <- c(1, 4, 2, 1, 4, 2)
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.25    2.00    2.33    3.50    4.00

> summary

function (object, ...)
UseMethod("summary")
<bytecode: 0x1c2f660>
<environment: namespace:base>
```

```
> methods(summary)

 [1] summary.aov             summary.aovlist
 [3] summary.aspell*         summary.connection
 [5] summary.data.frame      summary.Date
 [7] summary.default         summary.ecdf*
 [9] summary.factor          summary.glm
[11] summary.infl            summary.lm
[13] summary.loess*          summary.manova
[15] summary.matrix          summary.mlm
[17] summary.nls*            summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*
[21] summary.POSIXct         summary.POSIXlt
[23] summary.ppr*            summary.prcomp*
[25] summary.princomp*       summary.proc_time
[27] summary.srcfile         summary.srcref
[29] summary.stepfun         summary.stl*
[31] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

S3 dispatch

Since `summary()` is marked as a generic function, R is looking at `class()` of the first argument of `summary()` and based on it calls a function of name `summary.className()`. Since `summary.numeric()` does not exist, it calls `summary.default()`

```
> x <- c(1, 4, 2, 1, 4, 2)
> class(x)

[1] "numeric"

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.25    2.00    2.33    3.50    4.00

> summary.default(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    1.25    2.00    2.33    3.50    4.00
```

```
> y <- table(c(0, 1, 1, 0), c(1, 0, 1, 1))
> class(y)

[1] "table"

> summary(y)

Number of cases in table: 4
Number of factors: 2
Test for independence of all factors:
Chisq = 1.3, df = 1, p-value = 0.2
Chi-squared approximation may be incorrect

> summary.table(y)

Number of cases in table: 4
Number of factors: 2
Test for independence of all factors:
Chisq = 1.3, df = 1, p-value = 0.2
Chi-squared approximation may be incorrect
```

Methods: class-specific functions

Lets write the `id()` method that will return the sequence id. There are two parts to defining a class-specific function (**method**):

- Defining a generic function
- Defining the class method

```
> id <- function(x) {
+     UseMethod("id")
+ }  # generic
> id.GenericSeq <- function(x) {
+     x$id
+ }  # method
> id(s)

[1] "example dna sequence"
```

S3 methods mechanism

Generic function has the desired function name and contains only one command `UseMethod("functionName")` called a **dispatch command**. This command based on the first parameter's class calls an appropriate function of format `functionName.className`. If such function doesn't exist `functionName.default` is called.

The `seq()` method

Now consider the `seq()` function. This function already exists (try `?seq`). We would like to retain this old function, but also add our `seq()` that return the DNA/RNA sequence. The `seq()` function is **already a**

generic. We don't need to redefine it.

```
> seq

function (...)
UseMethod("seq")
<bytecode: 0x1a31620>
<environment: namespace:base>

> methods("seq")

[1] seq.Date    seq.default seq.POSIXt
```

Our `GenericSeq`-sepcific `seq` method

```
> seq.GenericSeq <- function(x) {
+     x$sequence
+ }
> seq(s)

[1] "AGCATACGACGACTACGACACTACGACATCAGACACTACAGACTACTACGACTACAC
```

S3 methods exercises

Look at the code we have written so far, understand it, and then solve the following exercise.

Exercise 1:

Explore some of the built-in generics and methods. Try the following commands:

```
methods("summary")
methods(class="lm")
```

Exercise 2: (code:`02_GenericSeq.R`, solution:`02_GenericSeq_solution.R`)

Both `length()` and `rev()` are already generic functions, but `alphabet()` is not. Add these methods for class `GenericSeq`:

- `length()` should return the length of the DNA/RNA sequence
- `alphabet()` should return the alphabet of the sequence
- `rev()` should return the sequence in reverse (Hint: try to use functions `strsplit()` and the existing base `rev()` function).

Reusing class methods

So far we have written methods for `GenericSeq` that work with any sequence type. Now lets introduce a new class `DnaSeq`. We want to **inherit** all methods from `GenericSeq` - to achieve this simply set the `class` attribute to all applicable class names.

```
> setSeqSubtype <- function(s) {
+     if (all(alphabet(s) %in% c("A", "C", "G", "T"))) {
+         class(s) <- c("DnaSeq", "GenericSeq")
+     } else if (all(alphabet(s) %in% c("A", "C", "G",
+         "U"))) {
+         class(s) <- c("RnaSeq", "GenericSeq")
+     } else {
+         stop("Alphabet ", alphabet(s), " is unknown.")
+     }
+
+     return(s)
+ }
> s.dna <- setSeqSubtype(s)
> class(s.dna)
```

S3 inheritance continued

DnaSeq methods

Define a DnaSeq method comp(). All GenericSeq methods still work with DnaSeq objects, but the comp() only works with DnaSeq.

```
> comp <- function(x) {
+     UseMethod("comp")
+ }
> comp.DnaSeq <- function(x) chartr("ACGT", "TGCA", seq(x))
```

S3 inheritance continued

```
> id(s)  # works on GenericSeq

[1] "example dna sequence"

> id(s.dna)  # works on DnaSeq, GenericSeq

[1] "example dna sequence"

> comp(s)  # fails with error

Error:  no applicable method for 'comp' applied to an
object of class "GenericSeq"

> comp(s.dna)

[1] "TCGTATGCTGCTGATGCTGTGATGCTGTAGTCTGTGATGTCTGATGATGCTGATGTC
```

S3 inheritance continued

S3 dispatch and inheritance

The dispatching will look for appropriate methods for all `x` (sub-)classes (in order in which they are set).

S3 inheritance exercise

Look at the inheritance code and understand how it works. Then solve the following exercise.

Exercise 3: (code: `03_inherit.R`, solution: `03_inherit_solution.R`)

Write the `comp()` method for `RnaSeq` class. Since we don't have a RNA FASTA file you will have to make a new `RnaSeq` object by hand and assign the right classes to test your code.

What do you notice about the S3 class system, is it easy to make mistakes? Could you also make your RNA sequence to be of class "`lm`"?

S3 class system revision

- Classes are implicit (no formal class definition)
- Making new objects is done by simply setting the `class` attribute
- Making class methods is done by defining a generic function `functionName()` and a normal function `functionName.className()`. Methods can be retrieved using the `methods()` function.
- Objects can inherit multiple classes by setting the `class` attribute to a vector of class names
- Many functions in base R use the S3 system
- Easy to make new ad-hoc classes and objects, but also mistakes and inconsistencies

The S4 class system is designed to address some of these concerns.

Plan

Differences of S4 class system to S3

- **Classes are explicit** - they have slots which describe what kind of data is stored
- **Improved introspection** - class, method and slot introspection
- **Consistency checking** - can no longer assign any class name, class hierarchy is explicitly checked and reinforced
- **Validity checking** - custom automatic checks of data consistency
- **Multiple inheritance, multiple dispatch, virtual classes**

S4 class system is the de-facto standard in Bioconductor.

Defining S4 class

Each class in S4 needs to be defined before it can be used. At this stage data types and inheritance are specified.

```
> setClass("GenericSeq",
+          representation = representation(
+            id = "character",
+            alphabet = "character",
+            sequence =  "character"
+           ))
```

S4 class slots

Slots define the **names and types** of variables that are going to be stored in the object. Types can be any of the basic R type or S3/S4 classes. To inspect how basic R types are called use `class()`, e.g.

```
> class("hello")

[1] "character"
```

Creating S4 objects

Once we have a class definition, we can make an object by filling out the slots. We can directly access the slots using the `@` notation although this is discouraged.

```
> genseq <- new("GenericSeq",
+               id="sequence name",
+               alphabet=c("A", "C", "G", "T"),
+               sequence="AGATACCCCGAAACGA")
```

S4 objects

```
> genseq

An object of class "GenericSeq"
Slot "id":
[1] "sequence name"

Slot "alphabet":
[1] "A" "C" "G" "T"

Slot "sequence":
[1] "AGATACCCCGAAACGA"

> genseq@id

[1] "sequence name"

> slot(genseq, "id")

[1] "sequence name"
```

Creating S4 methods

Similar to S3 we define object methods in two steps: by defining a **generic** and the **method**.

```
> setGeneric("rev", function(x) standardGeneric("rev"))

[1] "rev"

> setMethod("rev", "GenericSeq",
+           function(x)
+           paste(rev(unlist(strsplit(x@sequence, ""))),
+                 collapse=""))

[1] "rev"
```

```
> rev(genseq)

[1] "AGCAAAGCCCCATAGA"

> showMethods("rev")

Function: rev (package base)
x="ANY"
x="character"
    (inherited from: x="ANY")
x="GenericSeq"
```

S4 accessor methods

It is considered bad practice to use `@` in your code to access slots. It breaks the division between the internal class implementation and class usage.

Instead, create getter and setter methods for all slots you want to expose.

```
> setGeneric("id", function(object) standardGeneric("id"))

[1] "id"

> setMethod("id", "GenericSeq", function(object) object@id)

[1] "id"

> id(genseq)

[1] "sequence name"
```

```
> setGeneric("id<-",
+            function(object,value) standardGeneric("id<-"))

[1] "id<-"

> setReplaceMethod("id",
+                  signature(object="GenericSeq",
+                            value="character"),
+                  function(object, value) {
+                    object@id <- value
+                    return(object)
+                  })

[1] "id<-"

> id(genseq) <- "new sequence name"
> id(genseq)

[1] "new sequence name"
```

Exercise 4: (code: `04_basic_S4.R`)

Try the following introspection functions:

```
> showMethods("rev")
> getClass("GenericSeq")
> slotNames(genseq)
> getMethod("rev", "GenericSeq")
> findMethods("rev")
> isGeneric("rev")
```

What do these function output? In some cases the result is an object. Use the introspection functions to find out more about the results (e.g. `class()`, `getClass()`,...).

Exercise 5: (code as above, solution: `05_accessors_solution.R`)

Lets complete our `GenericSeq` implementation with some more methods. Implement getter/setter method `seq()` and getter only `alphabet()`. Then implement the method `length()` to return sequence length. First check if `"length"` is already a generic though.

Special methods - `show()`

You might have noticed that many object print a custom description instead of a plain list of slots. We can add this functionality by setting `show()` and `print()` methods.

```
> setMethod("show", "GenericSeq", function(object) {
+     cat("Object of class", class(object), "\n")
+     cat(" Id:", id(object), "\n")
+     cat(" Length:", length(object), "\n")
+     cat(" Alphabet:", alphabet(object), "\n")
+     cat(" Sequence:", seq(object), "\n")
+ })

[1] "show"

> genseq

Object of class GenericSeq
 Id: new sequence name
 Length: 16
 Alphabet: A C G T
 Sequence: AGATACCCCGAAACGA
```

Special methods - `print()`

The `print()` function already exists, but is not an S4 generic.

```
> setGeneric("print", function(x, ...) standardGeneric("print"))

[1] "print"

> setMethod("print", "GenericSeq", function(x) {
+     sq <- strsplit(seq(x), "")[[1]]
+     cat(">", id(x), "\n", " 1  ")
+     for (i in 1:length(x)) {
+         if ((i%%10) == 0)
+             cat("\n", i, " ")
+         cat(sq[i])
+     }
+     cat("\n")
+ })

[1] "print"
```

```
> print(genseq)

> new sequence name
  1   AGATACCCC
 10   GAAACGA
```

Special methods - `initialize()`

We might need to do some special processing on object creation. We can do this with a custom `initialize()` method.
Use named arguments with default values (otherwise class checking might fail).

```
> setMethod("initialize", "GenericSeq",
+     function(.Object, ..., id="", sequence="", alphabet=""){
+         .Object@id <- id
+         .Object@sequence <- toupper(sequence)
+         .Object@alphabet <- alphabet
+         callNextMethod(.Object, ...) # call parent class initialize()
+     })

[1] "initialize"

> show(new("GenericSeq", id="new seq.", alphabet=c("A", "T"), sequence="atatta"))

Object of class GenericSeq
 Id: new seq.
 Length: 6
 Alphabet: A T
 Sequence: ATATTA
```

Inheritance in S4 class system

Implementation of `GenericSeq` is finished. Now we want to re-use this code and add some extra functionality for `DnaSeq` and `RnaSeq`.

We start by defining the new classes that will inherit (contain) our `GenericSeq` class. It is good practise to provide some default (prototype) values.

```
> setClass("DnaSeq",
+          contains = "GenericSeq",
+          prototype = prototype(
+            id = paste("my DNA sequence", date()),
+            alphabet = c("A", "C", "G", "T"),
+            sequence = character()))
>
> setClass("RnaSeq",
+          contains = "GenericSeq",
+          prototype = prototype(
+            id = paste("my RNA sequence", date()),
+            alphabet = c("A", "C", "G", "U"),
+            sequence = character()))
```

Custom `comp()` methods in two subclasses

Now we can write the `comp()` method which is going to work differently for DNA and RNA sequences.

```
> setGeneric("comp", function(object, ...) standardGeneric("comp"))

[1] "comp"

> setMethod("comp", "DnaSeq", function(object, ...) {
+     chartr("ACGT", "TGCA", seq(object))
+ })

[1] "comp"

> setMethod("comp", "RnaSeq", function(object, ...) {
+     chartr("ACGU", "UGCA", seq(object))
+ })

[1] "comp"
```

Creating objects of appropriate class

We could use `new()` to create new object instances, but it is tedious and error prone. Instead, we should provide a function that reads in some data and sets the right class for the data.

```
> readFasta <- function(infile) {
+     lines <- readLines(infile)
+     header <- grep("^>", lines)
+     if (length(header) > 1) {
+         warning("Reading first sequence only.")
+         lines <- lines[header[1]:(header[2] - 1)]
+         header <- header[1]
+     }
+     .id <- sub("^> *", "", lines[header], perl = TRUE)
+     .sequence <- toupper(paste(lines[(header + 1):length(lines)],
+         collapse = ""))
+     .alphabet <- toupper(unique(strsplit(.sequence,
+         "")[[1]]))
+     if (all(.alphabet %in% c("A", "C", "G", "T"))) {
+         newseq <- new("DnaSeq", id = .id, sequence = .sequence)
+     } else if (all(.alphabet %in% c("A", "C", "G", "U"))) {
+         newseq <- new("RnaSeq", id = .id, sequence = .sequence)
+     } else {
+         stop("Alphabet ", .alphabet, " is unknown.")
+     }
+     return(newseq)
+ }
```

Object validity tests

The user can still use `new` in an inconsistent way or change a consistent object in the way that will render it inconsistent (e.g. assign an RNA sequence to an object of class `DnaSeq`). First lets make sure each new object is consistent, e.g. that alphabet matches sequence.

```
> setClass("GenericSeq",
+          representation = representation(
+            id = "character",
+            alphabet = "character",
+            sequence =  "character"),
+          validity = function(object) {
+            isValid <- TRUE
+            if (nchar(object@sequence)>0) {
+              chars <- casefold(unique(unlist(strsplit(object@sequence,""))))
+              isValid <- all(chars %in% casefold(object@alphabet))
+            }
+            if (!isValid)
+              cat("Some characters are not defined in the alphabet.\n")
+            return(isValid)
+          })
```

Validity tests - setters

Now lets make sure the user cannot render the objects inconsistent by modifying the object.

```
> setReplaceMethod("id", signature(object = "GenericSeq",
+     value = "character"), function(object, value) {
+     object@id <- value
+     if (validObject(object))
+         return(object)
+ })

[1] "id<-"


>
> setReplaceMethod("seq", signature(object = "GenericSeq",
+     value = "character"), function(object, value) {
+     object@sequence <- value
+     if (validObject(object))
+         return(object)
+ })

[1] "seq<-"
```

S4 exercises

Look at the code we wrote so far and understand it. Then solve the following exercise.

Exercise 6: (code: `06_S4_complete.R`)

Try again reading the supplied fasta file using

```
x <- readFasta("aDnaSeq.fasta")
```

Inspect the resulting object using object introspection tools. Try to break the resulting object by assigning invalid values to sequence. What happens if you do:

```
seq(x) <- "!"
```

and what if:

```
x@sequence <- "!"
```

Exercise 7: (code as above, solution: `07_transcribe_solution.R`)

Implement a new method `transcribe()` of `DnaSeq`. This method should take a `DnaSeq`, replace the T's with U's and return a `RnaSeq` object.

Virtual classes

A class can be marked to be **virtual** so that no objects can be made, but it can only be inherited. In our case, we might want to mark `GenericSeq` as virtual, to do so just add parameter `"VIRTUAL"` into class `representation`.

Class unions

In some cases we might want a slot to contain an object from one of multiple unrelated classes. In that case we would create a "dummy" class to serve as a place holder. For this we can use **class union**, for example `setClassUnion("AOrB", c("A", "B"))` would create a new virtual class `AOrB` that is a parent class to both `A` and `B`.

Overriding operators

Operators in R can also be overridden. For instance `setMethod("[", MyClass, ....)` will override the subsetting operator `[]` for `MyClass` to give it custom functionality. Other operators like `[[]]` and `$` can also be overridden.

```
> setMethod("[",
+           signature = signature(x = "GenericSeq",
+             i = "ANY", j = "missing"),
+           function(x, i, j, ..., drop = TRUE) {
+             paste(unlist(strsplit(x@sequence,""))[i], collapse="")
+           })

[1] "["

> genseq[1:10]

[1] "AGATACCCCG"
```

S4 generics clashes

Same generic name in two packages

What if two R packages, both using S4, provide a different generics definition?

```
> # generic in package1
> setGeneric("score", function(object, ...) standardGeneric("score"))
> # generic in package2
> setGeneric("score", function(x, ...) standardGeneric("score"))
```

Generics masking

Although the difference is only in the name of the argument we are dispatching on, the second generic is going to mask the first one, and any S4 methods that are defined for the first generic in package1 will no longer work.

Mutability in S3 and S4

Mutability

R objects are not **mutable**; R has a **pass-by-value** semantics, consistently with functional programming semantics. Whenever[a] you pass an object to a function, a copy is passed as argument; changes made to the object are local to the function call; the original object is unchanged. This is how things work for both S3 and S4 class systems.

[a] Although, in general, R tries to avoid copying objects unless they are modified.

```
> seq(a)

[1] "ACGTAA"

> comp(a)

[1] "TGCATT"

> seq(a)

[1] "ACGTAA"
```

Plan

Reference classes

- This paradigm uses **pass-by-reference** semantics: invoking a method may modify the content of the fields.
- Methods in this paradigm are associated with the object (rather than to generics)
- Java-like logic.
- See `?ReferenceClasses` for all the details.

Example

```
## here, you would have
> a$seq() ## equivalent of seq(a)
[1] "AGCATG"
> a$comp()
> a$seq()
[1] "TCGTAC"
```

Defining a reference class

Slots → a list of **fields**

```
> Seq <- setRefClass("Seq",
+                    fields = list(
+                      id = "character",
+                      alphabet = "character",
+                      sequence = "character"))
```

Generator objects

The return value of `setRefClass` is a **generator object** that is used to construct new object of given class.

Defining a reference class

Methods → a list of **functions**

```
> Seq <- setRefClass("Seq",
+                    fields = list(
+                      id = "character",
+                      alphabet = "character",
+                      sequence = "character"),
+                    methods = list(
+                      comp = function() {
+                        'Complements the (DNA) sequence' ## inline docs
+                        sequence <<- chartr("ACGT","TGCA",.self$sequence)
+                        id <<- paste(.self$id, "-- complemented")
+                      }
+                      ## there can be more, of course
+                      ))
```

Methods can be added either directly in class definition or later by calling `Seq$methods(functionName = function() { ...code... })`.

Using the reference classes

You also need to know that...

- accessing fields and calling methods is done with the $ operator.
- the current object can be referred to in a method by the reserved name `.self`.
- Changing fields of an object within methods needs to be done with the <<- operator.

```
> s <- Seq$new(id="foo", sequence="GATCATCA")
> s

Reference class object of class "Seq"
Field "id":
[1] "foo"
Field "alphabet":
character(0)
Field "sequence":
[1] "GATCATCA"

> s$sequence

[1] "GATCATCA"

> s$comp()
> s$sequence

[1] "CTAGTAGT"
```

Reference classes

Suitable for...

Reference classes are suitable for objects that are *dynamically tracked* by all the code: GUI components, read-only access to files (streams, data bases), internet resources, editing facilities, ...

Exercise 8: (code: `08_seqRefClass.R`)

We implemented some more methods using Reference Classes. Read through the methods and make sure you understand how they work. Then try out the test code in `08_seqRefClass.R`. What happens when we assign one Reference Class object to another?

Wrap up

- Object-oriented programming paradigm.
- S3 – easy, but can get unsafe; widely used in R.
- S4 – more verbose, but with more features (explicit classes, introspection, consistency and validity checks, multiple dispatch and inheritance, ...).
- Reference classes – pass-by-reference semantic, Java-like.