DSC 2003 Working Papers
(Draft Versions)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

The R.oo package - Object-oriented programming with references using standard R code

Henrik Bengtsson

Mathematical Statistics, Centre for Mathematical Sciences,
Lund University, Box 118, SE-221 00 Lund, Sweden.
hb@maths.lth.se

Abstract

When designing and implementing object-oriented applications in R, problems concerning generic functions and reference variables often occur. It is currently not clear how to create generic functions in a robust way such that a new package will be compatible with existing or future packages. This will become an important problem as more and more packages are made available. As a functional language, R does not provide methods for programming with references as all arguments to functions are copied by value. However, there are situations where it is useful or even necessary to pass arguments by references. In this paper we present the package R.oo, which overcomes these problems. In addition to a way for automatically creating generic functions in the background and a transparent way of using references, it also provides a foundation for designing and implementing object-oriented applications in a robust way. In this context, we also suggest a draft of a coding convention intended to bring additional structure to the source code. The package also extends the current exception handling mechanism in R such that exception objects can be caught based on their class.

Keywords: Object-oriented programming, generic functions, reference variables, coding conventions, exception handling, root class, S3, UseMethod.

1 Introduction

In R there are currently two ways for programming with classes, which are commonly referred to as the S3 (or S3/UseMethod) style [13] and the S4 style [7]. The

S3 style has been supported by R from the very beginning and support for the S4 style was added with the release of the methods package [9], which became part of the core distribution as of R v1.4.0 and is loaded by default from R v1.7.0. The package presented in this paper is currently relying exclusively on the S3 programming style, meaning that all classes defined using the package become S3 classes. However, future versions are likely to migrate to the S4 programming style. Having said this, it should be clear that the intention of the R.oo package is not to replace S3 or S4, but to extend them with a layer such that object-oriented design and programming in R become easier and more robust.

For beginners, but also for experts, it can be quite tedious to implement an object-oriented design in a robust way using the standard S3 (or S4) schema. It is easy to forget to add a generic function or, worse, to overwrite a non-generic function by a generic function. As a developer one has to stay up to date with all generic functions and all functions implemented in the latest R distribution to be sure that core functionalities in R are *not* overridden or removed when one's package is loaded. To be certain that the package works anywhere one must stay up to date with all other packages that the end user might load in parallel with your package. This approach is not scalable as more and more packages are added to the CRAN.[1]

In R arguments are supplied to functions by a *pass-by-value* semantic, also known as call-by-value. Inside the function, the arguments behave like local variables and any change made to a value inside the function is not reflected in the original value. The rational for this is that a function by definition takes one or several input values, returns something else and it should never modify the input values. When passing large data objects to a function a true pass-by-value approach would be too expensive in terms of time and memory. To overcome this, an argument in R is in practice *passed by reference* as long as the variable is not modified by the function. As soon as the function is changing the value of the argument, a local copy of it is created to obtain *pass by value*. Occasionally there are requests for adding a true *pass-by-reference* semantic so that functions *can* modify the original variable. From now on we refer to such functions as methods and reserve the word *function* for its original meaning. There are ways to emulate a pass-by-reference semantic by letting regular R variables represent reference variables or shortly *references*. With references it is possible to write more memory and time efficient code. Another advantage is that it is possible to write more user-friendly methods where fewer arguments need to be specified by the end user.

For reasons like these the R.oo package was developed. Its purpose is to relieve the developer from implementation details to make it possible to focus on the object-oriented design. The utility functions `setConstructorS3()` and `setMethodS3()` create constructor functions and class specific methods and at the same time make sure that required generic functions are created (or not) and throw exceptions if illegal method names, e.g. reserved words, are used etc. Furthermore, the package provides a totally transparent way of using *references* by defining a new "data type", namely the class *Object*. Any object of a class that inherits from this *root class* will be passed to functions as a reference. Using a single root class, which

[1] The current effort of adding *name spaces* [12] to the R language will remove some of the problems related to generic functions. A possible future support for *multiple generic function* is like to solve the problem completely.

all classes inherit from, improves the overall design and structure of any package developed. Finally, the R.oo package also provides an extended *exception handling* mechanism where an exception can be thrown and then caught depending on its class.

The R.oo package was written using standard R code ("100% R") and runs on all platforms. It was designed and implemented in a object-oriented style.
This document will *not* discuss what object-oriented programming and design are about. For an extensive case study on how to use the R.oo package see [3]. In section 2, we describe how to use classes that are defined using the R.oo package and that inherits from the root class Object. In this section some additional object-oriented features that come with the R.oo package are also described. To further improve the structure of an object-oriented implementation, we suggest an R Coding Convention (RCC) in section 3. The utility functions for defining constructors and methods, which are introduced in section 4, assert that the RCC is followed. The root class named Object will be described in detail in section 5 and the usage of references, which is provided by the Object class, will be discussed in section 6. Section 7 is about exception handling and section 8 explains some other utility functions that is part of the package. Other classes defined in the package are briefly mentioned in section 9. Section 10 explains how the package can be downloaded and installed. Conclusions are given in section 11.

2 Using classes

Throughout this document we make use of a simple case study example to describe the major parts of the package. Let the class SavingsAccount represent a bank account, which in the simplest case can be described by its balance. To secure against illegal modifications of the balance we represent the balance with a private field named `.balance` (see section 3). To obtain the balance of the account the function *getBalance()* is provided. Using *setBalance()* it is possible to modify the account balance directly, but it is not possible to set it to a negative balance. More commonly used are the methods for withdrawal and depositing, i.e. *withdraw(amount)* and *deposit(amount)*, respectively. The withdrawal method will not accept withdrawals if the balance becomes negative. The class inherits from the root class Object (section 5). When a SavingsAccount object is created, the balance will by default be set to zero. A Unified Modeling Language (UML) model of the SavingsAccount class is depicted in figure 1.

2.1 Creating an object

The implementation of the class is described in section 4, but for now assume that the usage of the constructor is `SavingsAccount(balance=0)` and that

```
account <- SavingsAccount(100)
```

creates a SavingsAccount object with initial balance 100. The object is referred to by the reference variable `account`.

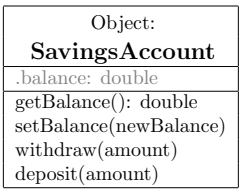

Figure 1: UML representation of the SavingsAccount class, which extends the Object class. Private fields has a . (period) as a prefix and 'Object:' in the header means that class extends the Object class.

2.2 Accessing fields

The fields of an instance of class inheriting from root class Object, shortly *an Object*, can be accessed similar to how elements of a list are accessed. For example, the balance field of the account object can the be retrieved by either `account$.balance` or `account[[".balance"]]`. To set the balance of the account either `account$.balance <- newBalance` or `account[[".balance"]] <- newBalance` will do.

Note that there are no ways to prevent the access to *private* fields. However, if one follows the RCC rule [4] that private fields and only private fields should have a . (period) prefix, it should be clear which fields should be accessed from outside and which should only be accessed from inside the SavingsAccount class. Moreover, private fields named this way will, by default, not be listed by the functions *getFields()* and *ll()*, which are described further in section 5, and neither by *ls()* [11].

2.3 Calling methods coupled with a class

Under the S3 schema a method coupled with a class is called in the same way as a regular function, but with (the reference to) the object as the first argument. When specifying the withdrawal and depositing methods above, we excluded the object argument for simplicity, e.g. *withdraw(amount)*. However, when calling the method one has to include it, e.g. `withdraw(account, amount)`. The method dispatching mechanism in S3/UseMethod will then make sure that the method of the correct class will be called. For more detailed information on how method dispatching is done in S3 see [11].

Similar to how fields are accessed, methods that are coupled with any Object derived class, can be accessed via the `$` (or the `[[`) operator, e.g. `account$withdraw(amount)`. In many other object-oriented languages such as Java and C++, but also the Omegahat's OOP project [8], this is the format used for methods calls. However, we do not recommend this style, except for static methods. Methods in R should be though of as belonging to generic functions [7] and not to a specific class per se.

2.4 Calling static methods

A *static method* of a class is invoked using only the class name and it does not require an instance of a class. All classes extending the Object class can define static methods. The most readable way to call a method of a class is via the `$` oper-

ator, e.g. `Object$load(file)` and `Exception$getLastException()` even though `load(Object(), file)` and `getLastException(Exception())` is also possible.

2.5 Accessing virtual fields

For Object instances, there is a third way of calling methods. Methods with a name of format `get<`*`Field`*`>(object)` or `set<`*`Field`*`>(object, value)` can be accessed by what we denote as *virtual fields*, i.e. as `object$<`*`field`*`>`. For instance the methods `getBalance(account)` and `setBalance(account, newBalance)` will be called whenever `account$balance` and `account$balance <- value` are evaluated, respectively.[2]

There are at least three real advantages of using virtual fields. First, it is possible, as the name suggest, to make it look like a class has a certain field, whereas it internally might use something else. For instance, a Circle class can have the two redundant fields named `radius` and `diameter` where one is a virtual field and the other is the actual field. Indeed, both might be declared virtual at the same time. We find that the use of virtual fields reduces the redundancy, which in turn reduces the risk for inconsistency. It also reduces the memory usage. Another advantage is that it is possible to restrict what values can be assigned to a field. For instance, we can prevent the user from setting a negative radius, e.g. `circle$radius <- -20`. Finally, virtual fields can prevent direct access to private fields or modification of constants, i.e. they provide a mechanism for *encapsulation* (data hiding).

2.6 Accessing class fields

A *class field*, also known as static fields, is a field associated with the class itself, not with a particular instance of the class. A class field of a class is shared by all objects of that class. A common role of a class field is that of a global variable (except from the important difference that it is not a global variable), e.g. `Colors$RED.HUE`. A class field is accessed as a regular field except that the object is now the static class object, e.g. `SavingsAccount$count <- SavingsAccount$count + 1`. Any class extending the Object class can have static fields.

3 Coding conventions

An important part of object-oriented design and implementation is to follow a standard for describing the design and for implementing it. There are several standards for describing object-oriented design of software, but one that has become the major standard is the Unified Modelling Language (UML) [10]. For implementation standards, also referred to as *coding conventions*, some languages have a well defined specification to follow whereas others do not. Unfortunately, there is no explicit and official coding convention for R. A well defined coding convention is useful because it helps to make the code more structured and more readable and it reduces the risk for mixing up field names with class names or reassign fields that are supposed

[2] By default, virtual fields have higher priority than regular fields, meaning that if virtual field exists that will be accessed first although it is possible on a reference-to-reference or an object-to-object basis to change the order which fields, virtual fields, methods and static methods are accessed.

to be constants etc. It is also fundamental for efficiently being able to share source code between developers and over time. For this reason we are working on a *R Coding Convention* (RCC) draft [4]. Next we will present an excerpt of its naming conventions.

3.1 Naming conventions

Some of the naming convention rules of the RCC apply to object-oriented design and programming. One of the most important is how classes, fields and methods should be named. According to RCC, names representing classes must be nouns and written in mixed case starting with upper case, e.g. `SavingsAccount`. Both field and method names must be in mixed case starting with lower case, e.g. `balance` and `getBalance()`. Private fields should have a . (period) as a prefix, e.g. `.balance`, to make it clear that it is a private field. Reserved keywords [11] and unsafe method names must also be avoided according to the RCC. The methods `setConstructorS3()` and `setMethodS3()`, described next, enforce these naming rules and if not followed, an `RccViolationException` is thrown. Not all rules are enforced to be backward compatible with some basic R functions that (for obvious reasons) do not comply with the RCC. As a last resort, it is always possible to turn of the test against RCC by using the argument `enforceRCC=FALSE` when using the above functions.

4 Defining new classes

Under the S3 schema there is no way to formally define a class and there is no way to enforce that an instance of a class has the correct format, contains the correct fields, or to assure that the inheritance structure is valid. One reason for this is that the class of the object and the inheritance structure of the class is solely specified by the class attribute of the individual objects. This attribute can be modified in any way at any time making the object-oriented implementation vulnerable to programming mistakes, but also misuse. The S4 schema over comes some of these lack-of-robustness drawbacks. The way the Object class is designed, the idea is that the fields (and hence the class) are defined inside the constructor function and the class attribute is never accessed by the programmer. This minimizes the risk for errors.

The two utility functions *setConstructorS3()* and *setMethodS3()* introduced next will help the programmer create constructors and methods without having to worry about generic functions. These functions can be used for defining any classes, not only classes derived from Object.

4.1 Defining constructors

The *setConstructorS3()* sets the constructor function and automatically creates any necessary generic function (there are cases where this might be necessary). When defining a class descending from the Object class, the constructor function also plays the role of defining the class (its fields) and specifying which class to extend. For example, to create the SavingsAccount class we write:

```
setConstructorS3("SavingsAccount", function(balance=0) {
```

```
  if (balance < 0)
    throw("Trying to create an account with a negative balance: ", balance);

  extend(Object(), "SavingsAccount",
    .balance = balance
  )
})
```

The declaration of the inheritance is done via the *extend()* method of the Object class, which will be called recursively throughout all the superclasses. The first argument to *extend()* should be the object returned by the constructor of the superclass. In the above example, the SavingsAccount inherits directly from the Object class, which is done by calling its constructor. The second argument to *extend()* should be the name of the class to be defined, e.g. SavingsAccount. According to the RCC, the name of the class should be the same as the name of the constructor function. Any other arguments to *extend()* are optional, but they must be named value arguments, e.g. `.balance=balance`, which then declare the fields of the class and their default values. Finally, all classes derived from Object *must* comply with the rule that it is must be possible to create an instance of it by calling its constructor with *no argument*[3], e.g. `account <- SavingsAccount()`, cf. *prototypes* in S4.

4.2 Defining methods

The *setMethodS3()* method creates methods for S3 classes and at the same time encapsulates a lot of details that the programmer should not have to think about. One such thing is if a generic function should be created or not and if so, how it should be created. For a detailed discussion on how generic functions are automatically created if missing see section 4.4. To create the *setBalance(newBalance)* method for the SavingsAccount class, the only thing needed is:

```
setMethodS3("setBalance", "SavingsAccount", function(this, newBalance) {
  if (newBalance < 0)
    throw("Trying to create an account with a negative balance: ", balance);
  this$.balance <- newBalance;
})
```

The complete usage of *setMethodS3()* is:

```
setMethodS3(name, class="default", definition, private=FALSE, protected=FALSE,
            static=FALSE, abstract=FALSE, trial=FALSE, deprecated=FALSE,
               envir=parent.frame(), createGeneric=TRUE, enforceRCC=TRUE)
```

where `name` is the name method, `class` is the name of the class and `definition` is the definition, i.e. the function, itself. If `class == "default"` (or `"ANY"`), a default function [11] is created, meaning that *setMethodsS3()* can be used whenever a function is defined. For all other arguments see the help page of the function.

[3]The reason for this is that *static class objects* are created by calling the constructor with no arguments.

4.3 Details

The *setMethodS3()* method creates a standard S3 method and at the same time makes sure that a generic function for that method is available. For instance, the evaluation of

```
setMethodS3("getBalance", "SavingsAccount", function(this) {
  this$.balance;
})
```

will create the S3 method for the class and the S3 generic function, i.e.

```
getBalance.SavingsAccount <- function(this) {
  this$.balance;
}

getBalance <- function(...) UseMethod("getBalance")
```

It also makes sure that if there already exists a non-generic function called `getBalance()`, then it will be renamed to `getBalance.default()`. If the latter already exists there is no way *setMethodS3()* can solve the conflict and therefore an Exception will be thrown explaining this. A generic function is not created if a generic function (or an internal function that works as such) already exists. For instance

```
setMethodS3("as.character", "SavingsAccount", function(this) {
  paste(data.class(this), ": balance is ", this$.balance, ".", sep="");
})
```

will only create the S3 method and *not* the generic function. In addition to this, *setMethodS3()* will by default assert that the (most important) RCC naming rules are followed. If not, it throws an RccViolationException informing that an RCC rule was violated.

4.4 Safely creating generic functions

When writing a package it is important to make sure that the package does not overwrite preexisting functions. If a preexisting function exists that is not a generic function, in most cases, the conflict can be solved by redefining the function to become a default function. The test whether a function already exists or not is commonly done manually by the programmer. However, new functions might be added when a new version of R is released and more seriously, other packages might be loaded before or after our package is loaded and there is no way we can know which functions will be defined or not. A much safer approach is to check for conflicts and solve them when the package is loaded. Furthermore, it is important to make sure that the generic function will work with all packages and not just the methods in our package. Complete object-oriented programming requires that methods can have the same name for different classes, but with another set of arguments. By not specifying the arguments of the generic functions, but only the special ... argument, e.g. `getArea <- function(...) UseMethod("getArea")` we the generic function is "as generic as possible"[4]. For a further discussion how to create generic functions safely see [5]. These problems are all taken care of automatically by *setMethodS3()*.

[4]If we *do* specify any arguments we restrict the corresponding methods for all classes in all packages loaded at the same time to have the exactly the same set of arguments. Under the S4

5 The root class Object

By enforcing that all classes are derived (directly or indirectly) from a common root class, we know that there exists a set of methods that are applicable to all such classes. This idea already exists to some extend in R, but using a common root class it will become more explicit to the end user. The R.oo package defines the root class *Object*, which has the following methods coupled to it. See also figure 2.

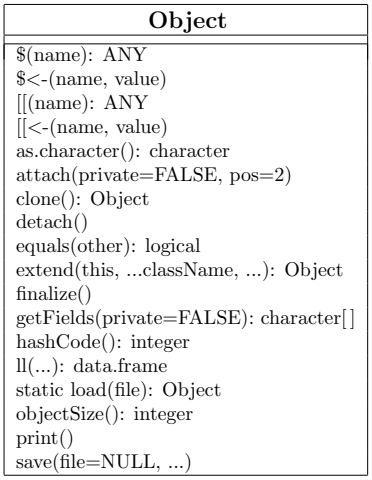

Figure 2: UML representation of the root class Object, which all classes should be derived from directly or indirectly through other classes. ANY is not a defined data type, but refers to any data type or class.

The method ***as.character()*** returns a string with short information about the Object. This is the same string that by default is displayed by *print()*.

The ***print()*** method prints information about the Object. By default the string returned by *as.character()* is printed. For convenience in R, *any* object of any data type or class whose name is typed at the command line followed by ENTER, the *print()* of that object will be called. Example:

```
> 1+2      # gives the object '3', which is then printed, i.e. print(3)
[1] 3
> account  # same as print(account)
SavingsAccount: balance is 100.
```

The method ***getFields(private=FALSE)*** returns the name of all fields in the Object. By default only names of non-private fields are returned.

The method ***ll(...)*** returns a data frame with detailed information about the fields of the Object. By default only non-private fields are listed. For more information

style, it is required and enforced that *all* methods for all classes have exactly the same arguments as the corresponding generic function. This is one of the reasons why we currently are not using the S4 style of programming with classes, but we hope to overcome this problem soon by making use of name spaces.

see section 8.

The ***hashCode()*** method returns an integer hash code for the Object.

The ***objectSize()*** method returns the (approximate) size of the Object.

The ***equals(other)*** method compares one Object with another. If they are equal the method returns `TRUE`, otherwise `FALSE`. If argument `other` is `NULL`, then `FALSE` is always returned. The default implementation of *equals()* is comparing the *hashCode()* values of both objects.

The ***clone()*** method creates an identical *copy* of the Object[5].

When an Object is deallocated from memory by the garbage collector the ***finalize()*** method is first called. Subclasses can override this method to make sure that any instances of those classes clean up after themselves. For instance, objects that allocate shared resources such as connections should make sure that these resources are closed and deallocated upon deletion.

The methods ***attach(private=FALSE, pos=2)*** and ***detach()*** attaches and detaches an Object to the search path, respectively. By default only public fields (private=FALSE) of an Object are attached and by default they are attached to the beginning of the search path (pos=2) just after the global environment. Any modification to such attached fields will *not* be reflected (saved) in the actual Object. The attach-detach mechanism should be used solely for read-only purposes.

The method ***save(file=NULL, ...)*** saves an Object to a file (or a connection) and the *static* method ***load(file)*** loads a previously saved Object and returns a reference to it.

The somewhat special method ***extend(...className, ...)*** extends an Object into a subclass named according to the string ...className[6] and which contains all fields as given by the ... arguments. This method is not intended to be overridden by any subclass.

Finally, as explained in the next section, the functionality for references is hidden inside the Object class. Hence, all subclasses will support references automatically and the programmer does not have to think about how reference variables should be implemented. They are always provided and they always behave in the same way.

6 Reference variables

All instances of the Object class or one of its subclasses are accessed via references variables or shortly *references*[7]. In standard R where reference variables are not

[5] Doing `ref2 <- ref` will only create a new reference to the same instance.

[6] The second argument to extend() has three dots as a prefix to make it possible to name fields as className or similar.

[7] For those who are not familiar with references but with pointers, references can be thought of as safe pointers to objects that can not by mistake be made to point to the wrong part of

provided, each instance of a class is accessed by one single variable, the object itself. With references, however, it is possible for several variables to access the same object. Here is an example where a list contains several references to the same Object:

```
person <- Person("Dalai Lama", 68)
l <- list(a=person, b=person, c=clone(person))
setAge(l$a, 67)
print(person)
[1] "Dalai Lama is 67 years old."
setAge(l$c, 69)
print(person)
[1] "Dalai Lama is 67 years old."
```

If `person` would *not* be a reference, the two elements `a` and `b` would be another two copies (clones) of the Person object and a modification of one of them would not have affected the other instance and neither the original variable `person`. It is possible to create a copy of an Object by using `clone()` as the above code shows.

Furthermore, references make it possible to implement software that would not otherwise be possible or would be very tedious to implement. Using references, more details can be encapsulated and thereby the package will be more user-friendly. We believe that a well designed object-oriented method interface based on references can serve as a base for, but also be a good complement to, a graphical user interface. For more complete real-world examples see [3] and [1].

6.1 Garbage collector

The use of references requires a memory management. Many languages, including R, provide a built-in *garbage collector*, which removes obsolete objects from the memory that are not referred to by anyone. Since objects inherited from the Object class are also standard R objects they will be recognized by the garbage collector. For example, an Object created inside a function and for which no reference is returned, will be deleted by the R garbage collector. In summary, objects does not have to be deleted explicitly, but for an Object to be deleted it is important that all references to it are removed, e.g. by *rm()*, or set to `NULL`. It is a good custom to do this as soon as an Object is not needed.

6.2 Details

R does *not* support references, but references can be emulated using so called *environments* [11]. However, using environments explicitly will quickly fill the source code with a lot of `get(name, envir=ref)`, `assign(name, value, envir=ref)` and/or `eval(..., envir=ref)` statements. This makes the code hard to read and increases the risk for errors. By *encapsulating* all calls to *get()* and *assign()* in the operator methods *$()*, *[[()*, *$<-()* and *[[<-()* of the root class Object, all fields can be accessed like if they were elements in a list. There are other ways for emulating references and some are more and some are less memory and time efficient than

the memory. Object-oriented programming where objects are passed by references does not differ much from the case when objects are passed by value. However, there are differences that are important to be aware of.

others. The R.oo package is using the first approach where each object lives in its own environment. One reason is that the garbage collector recognizes environment variables.

7 Exception handling

In addition to methods for defining classes and support for references, the package provides an improved *exception handling* mechanism. The core functionalities for exception handling is done by the Exception class (see figure 3). It provides methods to create and throw exceptions and together with its companion *trycatch()* complete exception handling is provided.

7.1 Creating and throwing exceptions

The easiest way to create and throw an exception is by calling *throw()*, e.g.

```
throw("Division by zero.")
```

which is equivalent to calling

```
throw(Exception("Division by zero."))
```

An object of any class that inherits from Exception contains information about the error and when it occurred. Any Exception object can be thrown using the *throw()* method and then optionally be caught by either *trycatch()* or *try()*. If an Exception is thrown, the last exception thrown can be obtained by the static method *getLastException()* of class Exception. The *as.character()* method for the

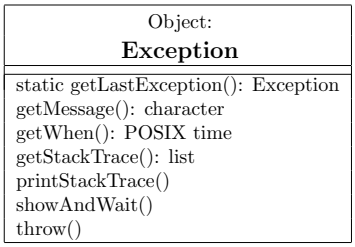

Figure 3: UML representation of the Exception class, which extends the Object class.

Object class is overridden by the Exception class and the default print message of an Exception has the format:

```
> throw("Division by zero.")
Error: [2002-10-20 10:24:07] Exception: Division by zero.
```

7.2 Catching exceptions depending on class

The *trycatch()* method can catch exceptions based on what class they belong to. Like the *try()* function, the first argument to *trycatch()* is the expression to be evaluated, which might throw an exception. Any further arguments must be named arguments where the name specifies the Exception class to be caught and the value

the code to be evaluated if such an exception is thrown. An argument with name `ANY` will catch any kind of Exceptions (including `try-error` thrown by `stop()`). If an exception is caught and no further exceptions are thrown, then `trycatch()` will return safely. The following code will generate and throw an exception, which will be caught by the `ANY` clause, preventing the R session from being interrupted.

```
trycatch({
  x <- log(2);
  y <- log("a");
}, ANY={
  x <- 0;
  y <- 0;
  print(Exception$getLastException());
})

print("trycatch() did indeed catch the exception.");
```

More over, code defined by an argument named `finally` is guaranteed to be evaluated immediately before *trycatch()* returns. This is for instance useful if a connection needs to be closed regardless of whether an exception is thrown or not.

8 Utility functions

In addition to the already mentioned methods, the package also defines some useful utility functions, which are applicable to objects of any class or data type. The default method of ***ll()*** lists detailed information about the objects (variables and functions) found in an environment. The returned data frame will by default contain information about the *member* (name of the variable or function), *data.class*, *dimension* and *object.size*, which are the values returned by the functions with the same.

```
> ll()
       member   data.class dimension object.size
1     analyze     function      NULL         248
2          ma       MAData         1         452
3         raw      RawData         1         452
4         gpr  GenePixData         1         460
5           y      numeric       100         828
```

For information about other utility functions provided by the package, see the help of the package.

9 Other classes

Other classes that are loaded with this package are *Class*, *Package* and *Rdoc*. The class Class provides an interface for querying classes about methods, fields etc. The class Package represents any kind of package, e.g. `Package("base")`. Given a Package object it is possible to query it for its classes, its author, check for updates (see below for an example) etc. The Rdoc class provides a compiler for Rdoc documentation, which is an extension of the Rd language that minimizes the

need for having to update the documentation when the source code is updated. For instance, the tag `@synopsis` generates a correct \`usage` (or a \`synopsis`) markup given the other information in the Rdoc code. The Rdoc documentation can be standalone files similar to Rd files (the simplest Rdoc file is a plain Rd file) or it can be part of the source files in form of comments. The Rdoc code is compiled into standard Rd files, which are then converted into help pages etc by `R CMD build`. We intend to extend the Rdoc compiler to recognize S4 classes too.

10 Installation

Since the package was written in 100% R, no native code needs to be compiled and the installation is straightforward. The R.oo package is part of a bundle of packages called R.classes [6]. To download and install the R.classes bundle do

```
install.packages("R.classes", contriburl="http://www.maths.lth.se/help/R")
```

from within R. By default the R library directory is the directory named `library/` in the directory where R is installed. The bundle can be installed in a private directory by setting the environment variable `R_LIBS`, e.g. `setenv R_LIBS $HOME/R/`. By loading the package, e.g. `library(R.oo)`, the correctness of the installation can be verified. To install on a Macintosh that does not have OS X or to manually install the bundle see [6]. For future updates, load the package and do

```
update(R.oo)
```

11 Conclusions

The R.oo package is open source, it is designed in an object-oriented style and implemented using plain and richly commented R code (100% R). More over, it is designed and implemented such that any future migration from S3 to S4 will be as smooth as possible for the end user.

For over two years we have been using the R.oo package and the R.classes bundle in a project developing a cDNA microarray analysis package (com.braju.sma [1]). We have found that by using the R.oo package we never have had problems with conflicts related to generic functions and we never have had to create a generic function explicitly. In cases when we by mistake tried to use a reserved word as method name, *setMethodS3()* immediately notified us. We have also found the Rdoc compiler to be a valuable tool for maintaining the nearly 200 Rd files. Due to the huge memory load and the large amount of redundancy in microarray data, the use of reference variables has been a natural and successful choice. More over, since methods can change the state of objects if references are used, we have been able to decrease the number of arguments that has to be specified in the method calls and therefore we can provide a cleaner and more user friendly method interface. For a further discussion how the R.oo package has been used in the development of our microarray package, see [2]. To install the com.braju.sma package see [1].

References

[1] Henrik Bengtsson. com.braju.sma - object-oriented microarray analysis in 100% R. http://www.maths.lth.se/help/R/, 2002.

[2] Henrik Bengtsson. The com.braju.sma package - a microarray analysis package based on an object-oriented design and reference variables. http://www.maths.lth.se/help/R/, 2002.

[3] Henrik Bengtsson. Programming with references - a case study using the R.oo package. http://www.maths.lth.se/help/R/, 2002.

[4] Henrik Bengtsson. R Coding Conventions (draft). http://www.maths.lth.se/help/R/, 2002.

[5] Henrik Bengtsson. Safely creating S3 generic functions using setGenericS3(). http://www.maths.lth.se/help/R/, 2002.

[6] Henrik Bengtsson. The R.classes bundle (R.oo and friends). http://www.maths.lth.se/help/R/, 2003.

[7] John M. Chambers. *Programming with Data*. Springer, 1998.

[8] John M. Chambers. OOP programming in the S language. http://www.omegahat.org/, 2002.

[9] John M. Chambers. S language methods and classes. http://www.omegahat.org/RSMethods/, 2002.

[10] Object Management Group. UML Resource Page. http://www.omg.org/uml/, 2002.

[11] R Development Core Team. R Language Definition (v1.7.0, draft). http://www.r-project.org/, April 2003.

[12] R Development Core Team. Writing R Extensions (v1.7.0). http://www.r-project.org/, April 2003.

[13] W.N. Venables and B.D. Ripley. *Modern Applied Statistics with S-PLUS*. Springer, 3rd edition, 1999.