

R's Object Oriented Programming System: S3

Stats 102A

Miles Chen

Department of Statistics and Data Science



Section 1

Object-Oriented Programming

Object-Oriented Programming

Object-oriented programming (OOP) is a style of programming that focuses on defining different types of objects and functions that can be applied to those object types.

OOP in R can be challenging due to the existence of four distinct systems - S3, S4, RC, and R6 - each with different design philosophies and trade-offs.

People have different opinions about which OOP systems are important in R. I am heavily influenced by Hadley Wickham's teachings on the subject.

In this class, we will focus on S3 (widely used and the backbone of many base R functions) and R6 (which supports encapsulated, reference-based programming), as these systems are most relevant for our purposes.

OOP Paradigms

In R, the S3 and S4 systems implement OOP through **generic functions**, where functions inspect the object's class to decide which method to execute.

While both S3 and S4 rely on generic functions, S3 is more informal and relies on naming conventions, whereas S4 provides formal class definitions and method signatures, including built-in validity checks.

The RC and R6 systems support **encapsulated** OOP, where methods are bundled with the data, and objects are mutable (i.e., they can be modified in place).

Regardless of the system, all OOP implementations in R use objects to store data and associated methods to perform actions on that data.

Encapsulated OOP

Many popular programming languages (e.g., Python, C++, Java) use encapsulated OOP, where each object (or class) encapsulates its data and methods.

In encapsulated OOP, you typically call a method with syntax such as `object$method(arg1, ...)` (as in R6), which emphasizes that the method belongs to the object.

You can think of a “camera” object. The camera has fields to keep information like how many pictures have been taken, how much memory is available. The camera has methods (verbs) it can perform: shoot a photo, focus, record a video.

Encapsulated OOP is available in R via **R6** and **RC**

Generic Function OOP

In generic function OOP, methods are defined as part of generic functions rather than being attached directly to objects. The objects themselves only contain data (often via a class attribute), and the appropriate method is chosen at runtime based on the object's class.

This design enables polymorphism, where a single generic function (such as `print()`, `summary()`, or `plot()`) can behave differently depending on the class of the object it is called with.

You can think of the verb “shoot.” This verb takes different meanings or behaviors based on the object we are talking about. If you have a camera, *shoot* means take a picture. If I give you a hockey puck, *shoot* means hitting it towards the goal. If there's a gun, *shoot* means pulling the trigger. With “the breeze,” the word *shoot* means to chat idly. In generic function OOP, how the function (verb) behaves depends on the object.

A **polymorphic** function is one that behaves differently for different input types. The **S3** and **S4** systems in R use generic polymorphic functions. In base R, all object oriented behaviors are done in the **S3** system.

Example of a Polymorphic Function

Consider the `summary()` function, which adapts its output depending on the type of input.

```
library(ggplot2)
summary(diamonds$carat)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.2000	0.4000	0.7000	0.7979	1.0400	5.0100

```
summary(diamonds$cut)
```

Fair	Good	Very Good	Premium	Ideal
1610	4906	12082	13791	21551

In this example, the `summary()` function dispatches different methods based on whether it is summarizing a numeric variable (`carat`) or a categorical variable (`cut`), illustrating the principle of polymorphism in generic function OOP.

Classes and Methods

Central to any object-oriented system are the concepts of **classes** and **methods**.

- A **class** defines the structure of an object. Typically, a class specifies several **fields** (or properties) that hold data relevant to that object.
- A **method** is a function that implements behavior for objects of a specific class.

In essence, a class defines what an object is, while methods define what the object can do.

Inheritance is a key feature of many object-oriented languages. In an inheritance hierarchy, a class (the child or subclass) can inherit properties and methods from another class (the parent or superclass). If a method is not defined for a child class, the parent's method is used instead. We say that the child **inherits** behavior from its parent.

Example:

- An ordered factor may inherit from a regular factor.
- A tibble (the tidyverse take on data frames) inherits from the base data frame class.

Method Dispatch and Generic Functions

Method dispatch is the process of selecting the correct method for an object based on its class.

In R, **generic functions** (or **generics**) use method dispatch. When a generic function (such as `print()`, `summary()`, or `plot()`) is called, it checks the class of its argument(s) and selects the appropriate method to execute.

OOP Systems in R

• S3

- ▶ R's earliest and simplest object-oriented system.
- ▶ It is informal, relying on naming conventions rather than formal class definitions.
- ▶ S3 is used extensively in the base and stats packages and is the most common system in many CRAN packages.

• S4

- ▶ A more formal and rigorous system than S3.
- ▶ It requires explicit definitions of classes, methods, and generic functions, and supports formal method signatures and validity checks.
- ▶ S4 is implemented in the `methods` package.

• RC (Reference Classes)

- ▶ Implements a form of encapsulated OOP using message-passing.
- ▶ In RC, methods are part of the class definition, and objects are mutable (they can change state).

• R6

Besides the object-oriented systems, R has **base types**, which are the underlying C-level types that form the building blocks of all R objects.

- **Base types** include atomic vectors, lists, functions, environments, and more exotic objects (such as names, calls, and promises).
- They are not part of an OOP system because new base types can only be defined by the R core team and are rarely added.
- You can inspect an object's base type using the `typeof()` function.

Base Types

Underlying every R object is a C structure (a struct) that stores:

- The object's contents,
- Information required for memory management, and
- A **type**, which is the object's **base type**.

While base types provide the fundamental building blocks for R objects, they are not themselves object-oriented because they lack a class attribute. Most user-level OOP in R is built on top of these base types.

You can check if an object is “pure” (i.e., not an object-oriented instance) by verifying that `is.object(x)` returns `FALSE`.

Inconsistent Naming of Base Types

It's important to note that base types in R sometimes have inconsistent naming conventions. The name returned by `typeof()` might not match the name used in the corresponding `is.*` function. For example:

```
typeof(mean)
```

```
[1] "closure"
```

```
is.function(mean)
```

```
[1] TRUE
```

```
typeof(sum)
```

```
[1] "builtin"
```

```
is.primitive(sum)
```

```
[1] TRUE
```

The `is.object()` Function

Functions that behave differently for different base types are almost always written in C. Even if you never write C code, it is important to understand base types, since everything else is built on top of them:

- S3 objects can be built on top of any base type
- S4 objects use a special base type
- RC objects are a combination of S4 and environments (another base type).

The basic difference between base and object-oriented objects is that object-oriented objects have a class attribute.

To see if an object is a pure base type (i.e., does not have a class attribute), check that `is.object(x)` returns `FALSE`.

Section 2

The S3 Object-Oriented System

The S3 Object-Oriented System

S3 is R's first and simplest object-oriented system.

Most objects that you encounter are S3 objects. Unfortunately, there is no simple way to test if an object is an S3 object in base R.

The closest you can come is `is.object(x) & !isS4(x)`, i.e., it is an object but not an S4 object.

The `is.object()` Function

```
df <- data.frame(x = 1:10, y = letters[1:10], stringsAsFactors = TRUE)  
is.object(df)
```

```
[1] TRUE
```

```
isS4(df)
```

```
[1] FALSE
```

```
is.object(df$x)
```

```
[1] FALSE
```

```
is.object(df$y)
```

```
[1] TRUE
```

Generic Functions and UseMethod()

In S3, methods belong to functions, called **generic functions**, or **generics** for short. S3 methods do not belong to objects or classes. (In contrast, most OOP systems in other programming languages use **encapsulated OOP**, in which methods belong to objects/classes. S3 uses **functional OOP**, in which methods belong to functions.)

To determine if a function is an S3 generic, you can inspect its source code for a call to `UseMethod()`, which is the function that figures out the correct method to call, the process of **method dispatch**.

```
mean
```

```
function (x, ...)
  UseMethod("mean")
<bytecode: 0x0000022731230580>
<environment: namespace:base>
```

Some S3 generics, like `[`, `sum()`, and `cbind()`, do not call `UseMethod()` because they are implemented in C. Instead, they call the C functions `DispatchGroup()` or `DispatchOrEval()`.

Functions that do method dispatch in C code are called **internal generics** and are documented in `?"internal generic"`.

Recognizing S3 Methods

Given a class, the job of an S3 generic is to call the right S3 method. You can recognize S3 methods by their names, which look like `generic.class()`.

For example, the Date method for the `mean()` generic is called `mean.Date()`, and the factor method for `print()` is called `print.factor()`.

Recognizing S3 Methods

The `generic.class()` method syntax is the reason that most modern style guides discourage the use of `.` in function names: it makes them look like S3 methods.

For example, is `t.test()` the `t` method for objects that are of class `test`?

Similarly, the use of `.` in class names can also be confusing: is `print.data.frame()` the `print()` method for `data.frame` class objects, or the `print.data()` method for `frame` class objects?

To reduce confusion, most style guides prefer the underscore `_` instead of the `.`, as in `read_csv()` or `as_tibble()`.

Generic Function Example

The polymorphic behavior of generic functions allows you to use the same function name on different object types.

For example, the function `t()` can be used on a matrix:

```
x <- matrix(1:12, nrow = 4)
t(x)
```

	[,1]	[,2]	[,3]	[,4]
[1,]	1	2	3	4
[2,]	5	6	7	8
[3,]	9	10	11	12

Generic Function Example

The `t()` function can also be used on a data frame:

```
df <- data.frame(a = 1:4, b = 5:8, c = 9:12)
t(df)
```

	[,1]	[,2]	[,3]	[,4]
a	1	2	3	4
b	5	6	7	8
c	9	10	11	12

The functions behave similarly, which is important for the user experience. However, the internal code for each function is different.

Internatl code of t()

```
t.default
```

```
function (x)
  .Internal(t.default(x))
<bytecode: 0x00000227353b9000>
<environment: namespace:base>
```

```
t.data.frame
```

```
function (x)
{
  x <- as.matrix(x)
  NextMethod("t")
}
<bytecode: 0x0000022735931d58>
<environment: namespace:base>
```

When you apply `t()` to a `data.frame` object, it will call the function `t.data.frame()`, which first converts the `data.frame` into a matrix and then applies the next method available for the object (which will be the default method: `t.default()`).

The methods() Function

You can see all the methods that belong to a generic with `methods()`:

```
methods("mean")
```

```
[1] mean,ANY-method          mean,denseMatrix-method  mean,sparseMatrix-  
method  
[4] mean,sparseVector-method mean.Date                mean.default  
[7] mean.difftime            mean.POSIXct             mean.POSIXlt  
[10] mean.quosure*             mean.vctrs_vctr*  
see '?methods' for accessing help and source code
```

```
methods("t.test")
```

```
[1] t.test.default* t.test.formula*  
see '?methods' for accessing help and source code
```

The methods() Function

You can also list all generics that have a method for a given class:

```
methods(class = "ts") # methods available for time series objects
```

```
[1] [          [<-      aggregate    as.data.frame as_tibble
[6] cbind      cbind2      coerce      cycle         diff
[11] diffinv    filter      head        initialize    kernapply
[16] kronecker  lines       Math        Math2         monthplot
[21] na.omit    Ops         plot        print         rbind2
[26] show       slotsFromS3 t           tail          time
[31] window     window<-
```

see '?methods' for accessing help and source code

Section 3

Defining S3 Classes

Defining Classes and Creating Objects

S3 is a simple and ad hoc system in the sense that it has no formal definition of a class.

To make an object an instance of a class, you only need to set the **class** attribute for a base object.

You can do that during creation with `structure()`, or after the fact with `class<=()`:

```
# Create and assign class in one step
x <- structure(list("apple"), class = "fruit")

# Create, then set class
y <- list("banana")
class(y) <- "fruit"
```

The above code has just defined a new class of S3 object called “fruit”

The `class()` and `inherits()` Functions

S3 objects are usually built on top of lists or atomic vectors with attributes. Functions can also be turned into S3 objects.

Other base types are either rarely seen in R or have unusual semantics that do not work well with attributes.

You can determine the class of an S3 object with the `class()` function. You can see if an object inherits from a specific class using `inherits(x, "classname")`.

```
class(x)
```

```
[1] "fruit"
```

```
inherits(x, "fruit")
```

```
[1] TRUE
```

Multiple Classes

The class of an S3 object can be a vector, which describes behavior from most to least specific.

For example, the class of the tibble object is a vector `c("tbl_df", "tbl", "data.frame")` indicating that tibbles inherit behavior from data frames.

```
class(diamonds)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

Ordered factors are also examples of objects with multiple classes.

```
class(diamonds$cut)
```

```
[1] "ordered" "factor"
```

`inherits()` vs `class()` == "classname"

When checking to see if an object is of a certain class, use `inherits(object, "classname")` rather than `'class(object) == "classname"'`

```
inherits(diamonds$cut, "factor") # returns a single value
```

```
[1] TRUE
```

```
class(diamonds$cut) == "factor" # returns a vector because class has length 2
```

```
[1] FALSE TRUE
```

Most S3 classes provide a **constructor** function that creates new objects with the correct structure.

The constructor should follow two principles:

- Have one argument for the base object, and one for each attribute.
- Check the type of the base object and the types of each attribute.

Constructor functions usually have the same name as the class, much like the built-in ones (e.g., `factor()` and `data.frame()`).

Constructors

An example of a constructor:

```
fruit <- function(x) {  
  stopifnot(is.character(x)) # checks to see if x is a character vector  
  structure(list(x), class = "fruit")  
}  
# in use:  
z <- fruit("pineapple")  
z
```

```
[[1]]  
[1] "pineapple"  
  
attr(,"class")  
[1] "fruit"
```

Changing Classes

Aside from built-in constructor functions, S3 has no checks for correctness. This means you can change the class of existing objects:

```
# Create a linear model
lm_mtcars <- lm(log(mpg) ~ log(displacement), data = mtcars)
class(lm_mtcars)
```

```
[1] "lm"
```

```
print(lm_mtcars)
```

Call:

```
lm(formula = log(mpg) ~ log(displacement), data = mtcars)
```

Coefficients:

(Intercept)	log(displacement)
5.3810	-0.4586

Changing Classes

```
# Change the class to data.frame
class(lm_mtcars) <- "data.frame"
print(lm_mtcars) # No longer prints properly
```

```
[1] coefficients residuals effects rank fitted.values
[6] assign qr df.residual xlevels call
[11] terms model
<0 rows> (or 0-length row.names)
```

```
# But the data is still inside
lm_mtcars$coefficients
```

```
(Intercept) log(displacement)
5.3809725 -0.4585683
```

Changing Classes

The lack of built-in validation of classes has the potential to be problematic, but it rarely causes issues in practice.

While you *can* change the class of an object, you never should. R does not protect you from yourself.

Section 4

Creating S3 Methods and Generics

Creating S3 Generics

The job of an S3 generic is to perform method dispatch, i.e. find the specific implementation for a class.

To create a new generic, create a function that calls `UseMethod()`.

The `UseMethod()` function takes two arguments: the name of the generic function, and the argument to use for method dispatch. The second argument defaults to the first argument of the function, which is usually what you want.

```
quotation <- function(x) {  
  UseMethod("quotation")  
}
```

There is no need to pass any of the arguments of the generic to `UseMethod()`. The `UseMethod()` function will pass arguments to the method automatically (using what Hadley Wickham refers to as “deep magic”).

Creating S3 Methods

A generic is not useful without some methods.

To add a method to a generic, create a regular function with the correct (`generic.class`) name:

```
quotation.fruit <- function(x) {  
  "Fruits are an important part of a balanced diet."  
}
```

```
x <- structure(list("banana"), class = "fruit")  
class(x)
```

```
[1] "fruit"
```

```
quotation(x)
```

```
[1] "Fruits are an important part of a balanced diet."
```

Adding Methods to Existing Generics

Adding a method to an existing generic works in the same way:

```
mean.fruit <- function(x) {  
  5  
}
```

```
mean(x)
```

```
[1] 5
```

As we saw previously with changing classes, there is no check to make sure that the method returns the class compatible with the generic. It is up to you to make sure that your method does not violate the expectations of existing code.

Method Dispatch

How does method dispatch work?

The `UseMethod()` function creates a vector of function names, like `paste0("generic", ".", c(class(x), "default"))` and looks for each potential method in turn.

The “**default**” class makes it possible to set up a fallback method for otherwise unknown classes.

The “default” class is a special **pseudo-class** in that it is not a real class, but we can define methods for it to use for inputs with unknown classes.

Method Dispatch Example

Consider the following generic and methods:

```
quotation <- function(x) {  
  UseMethod("quotation")  
}  
quotation.fruit <- function(x) {  
  "Fruits are an important part of a balanced diet."  
}  
quotation.apple <- function(x) {  
  "An apple a day keeps the doctor away."  
}  
quotation.default <- function(x) {  
  "The default quotation: Let food be thy medicine and medicine be thy food."  
}
```

Method Dispatch Example

```
# Dispatches method for apple class
a <- structure(list("Fuji"), class = c("apple", "fruit"))
quotation(a)
```

```
[1] "An apple a day keeps the doctor away."
```

```
# No method for banana class, so uses method for fruit class
b <- structure(list("Chiquita"), class = c("banana", "fruit"))
quotation(b)
```

```
[1] "Fruits are an important part of a balanced diet."
```

```
# No method for donut class, so falls back to default
c <- structure(list("Dunkin"), class = "donut")
quotation(c)
```

```
[1] "The default quotation: Let food be thy medicine and medicine be thy food."
```

Methods Are Functions

Because methods are normal R functions, they can also be called directly:

```
b <- structure(list("Chiquita"), class = c("banana", "fruit"))  
# Call the correct method with dispatch:  
quotation(b)
```

```
[1] "Fruits are an important part of a balanced diet."
```

```
# Force R to call the wrong method:  
quotation.apple(b)
```

```
[1] "An apple a day keeps the doctor away."
```

However, this is just as dangerous as changing the class of an object, so you should not do it.

Method Dispatch

If there is no default method and you attempt to use a function on an object for which no method exists, R will throw an error.

```
rm(quotation.default) # we remove the default method
c <- structure(list("Dunkin"), class = "donut")
quotation(c) # we call the function on an object with class donut
```

Error in `UseMethod()`:

! no applicable method for 'quotation' applied to an object of class "donut"

Section 5

Inheritance in S3 Classes

Inheritance in S3 Classes

S3 classes can share behavior through **inheritance**.

Inheritance in S3 follows three principles:

- The class of an object can be a character vector.
- If a method is not found for the class in the first element of the vector, R looks for a method for the second class (and so on).
- A method can delegate work by calling `NextMethod()`.

We will expand on these principles in this section.

Method Dispatch Hierarchy

The `s3_dispatch()` function in the `sloop` package inputs a function call and outputs the list of all possible function names that are considered for method dispatch.

- The lack of a symbol means the method does not exist.
- The `=>` arrow means the method exists and is found by `UseMethod()`.
- The `->` arrow means the method exists and is used through `NextMethod()`.
- The `*` means the method exists but is not used.

Method Dispatch Hierarchy

For example, consider calling the generic `print()` function on an ordered factor:

```
library(sloop)
s3_dispatch(print(factor(letters, ordered = TRUE)))
```

```
    print.ordered
=> print.factor
   * print.default
```

There is no `print()` method for the ordered class, so the `print()` generic dispatches the `print.factor()` method for ordered objects.

Subclasses and Superclasses

The ordered class is said to be a **subclass** of factor because it always appears before it in the class vector.

Conversely, the factor class is a **superclass** of ordered.

S3 has no formal restrictions on the relationship between sub- and superclasses, but there are two principles to follow when creating a subclass:

- The base type of the subclass should be that same as the superclass.
- The attributes of the subclass should be a superset of the attributes of the superclass.

Section 6

Method Dispatch Self-Quiz

Method Dispatch Self-Quiz

The following slides can be seen as a self-quiz for the topic of method dispatch.

The functions and methods are very simple, usually adding a constant like 2 or 10. Because they are so simple, I've opted to leave them as a single line.

With each slide, try to predict the output. The full details of what methods R searches are provided with the `s3_dispatch()` function.

As far as studying, you need to learn which method will be dispatched. You do not need to learn the exact output of the `s3_dispatch()` function itself.

Method Dispatch Self Quiz

```
rm(list = ls())  
f  <- function(x) UseMethod("f") # the generic f function  
f.j <- function(x) x + 2  # the f method for class j  
f.k <- function(x) x + 10 # the f method for class k  
k <- 1
```

Method Dispatch Self Quiz

```
rm(list = ls())  
f  <- function(x) UseMethod("f") # the generic f function  
f.j <- function(x) x + 2  # the f method for class j  
f.k <- function(x) x + 10 # the f method for class k  
k <- 1
```

```
f(k)
```

Method Dispatch Self Quiz

```
rm(list = ls())  
f  <- function(x) UseMethod("f") # the generic f function  
f.j <- function(x) x + 2  # the f method for class j  
f.k <- function(x) x + 10 # the f method for class k  
k <- 1
```

```
f(k)
```

Error in `UseMethod()`:

! no applicable method for 'f' applied to an object of class "c('double', 'numeric')"

```
s3_dispatch(f(k)) # full details of the result
```

```
f.double  
f.numeric  
f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "j"  # object k has a class of "j"
f(k)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "j"  # object k has a class of "j"
f(k)
```

```
[1] 3
attr(,"class")
[1] "j"
```

```
s3_dispatch(f(k))
```

```
=> f.j
    f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "k"  # object k has a class of "k"
f(k)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "k" # object k has a class of "k"
f(k)
```

```
[1] 11
attr(,"class")
[1] "k"
```

```
s3_dispatch(f(k))
```

```
=> f.k
    f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "k"
f.default <- function(x) x + 100
f(k)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
k <- 1
```

```
class(k) <- "k"
f.default <- function(x) x + 100
f(k)
```

```
[1] 11
attr(,"class")
[1] "k"
```

```
s3_dispatch(f(k)) # full details of the result
```

```
=> f.k
* f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

```
class(k) <- c("a","b")
f(k)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

```
class(k) <- c("a","b")
f(k)
```

```
[1] 101
attr("class")
[1] "a" "b"
```

```
s3_dispatch(f(k))
```

```
f.a
f.b
=> f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

```
class(k) <- NULL
f(k)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
k <- 1
```

```
class(k) <- NULL
f(k)
```

```
[1] 101
```

```
s3_dispatch(f(k))
```

```
  f.double
  f.numeric
=> f.default
```

Self Quiz

```
f    <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
l <- structure(10, class = c("k", "l"))
f(l)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
l <- structure(10, class = c("k", "l"))
f(l)
```

```
[1] 20
attr("class")
[1] "k" "l"
```

```
s3_dispatch(f(l))
```

```
=> f.k
* f.l
* f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","l")
f(l)
```

Self Quiz

```
f <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2 # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","l")
f(l)
```

```
[1] 60
attr(,"class")
[1] "m" "l"
```

```
s3_dispatch(f(l))
```

```
  f.m
=> f.l
  * f.default
```

Self Quiz

```
f    <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","n")
f(l)
```

Self Quiz

```
f <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2 # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","n")
f(l)
```

```
[1] 110
attr(,"class")
[1] "m" "n"
```

```
s3_dispatch(f(l))
```

```
f.m
f.n
=> f.default
```

Self Quiz

```
f    <- function(x) UseMethod("f") # the generic f function
f.j  <- function(x) x + 2  # the f method for class j
f.k  <- function(x) x + 10
f.default <- function(x) x + 100
f.l  <- function(x) x + 50
l    <- 10
```

Self Quiz

```
f    <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","n")
f.j(l)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
l <- 10
```

```
class(l) <- c("m","n")
f.j(l)
```

```
[1] 12
attr(,"class")
[1] "m" "n"
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
f(7)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
f(7)
```

```
[1] 107
```

```
s3_dispatch(f(7))
```

```
  f.double
  f.numeric
=> f.default
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
f.j(7)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
```

```
f.j(7)
```

```
[1] 9
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

```
f(7)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

```
f(7)
```

```
[1] 107
```

```
s3_dispatch(f(7))
```

```
  f.double
  f.numeric
=> f.default
```

Self Quiz

```
f    <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

```
f(7L)
```

Self Quiz

```
f  <- function(x) UseMethod("f") # the generic f function
f.j <- function(x) x + 2  # the f method for class j
f.k <- function(x) x + 10
f.default <- function(x) x + 100
f.l <- function(x) x + 50
f.integer <- function(x) 100 * x
```

```
f(7L)
```

```
[1] 700
```

```
s3_dispatch(f(7L))
```

```
=> f.integer
    f.numeric
    * f.default
```

Self Quiz

```
# Note the name in UseMethod is "g" instead of "f". You should never do this.
f  <- function(x) UseMethod("g")
f.j <- function(x) x + 2
g.j <- function(x) -1 * (x + 2)
f.default <- function(x) x + 100
g.default <- function(x) -1 * (x + 100)
```

Self Quiz

```
# Note the name in UseMethod is "g" instead of "f". You should never do this.
f  <- function(x) UseMethod("g")
f.j <- function(x) x + 2
g.j <- function(x) -1 * (x + 2)
f.default <- function(x) x + 100
g.default <- function(x) -1 * (x + 100)

k <- structure(10, class = "j")
f(k)
```

Self Quiz

Note the name in UseMethod is "g" instead of "f". You should never do this.

```
f <- function(x) UseMethod("g")
f.j <- function(x) x + 2
g.j <- function(x) -1 * (x + 2)
f.default <- function(x) x + 100
g.default <- function(x) -1 * (x + 100)
```

```
k <- structure(10, class = "j")
f(k)
```

```
[1] -12
attr(,"class")
[1] "j"
```

Self Quiz

```
# Note the name in UseMethod is "g" instead of "f". You should never do this.  
f  <- function(x) UseMethod("g")  
f.j <- function(x) x + 2  
g.j <- function(x) -1 * (x + 2)  
f.default <- function(x) x + 100  
g.default <- function(x) -1 * (x + 100)
```

Self Quiz

```
# Note the name in UseMethod is "g" instead of "f". You should never do this.
f  <- function(x) UseMethod("g")
f.j <- function(x) x + 2
g.j <- function(x) -1 * (x + 2)
f.default <- function(x) x + 100
g.default <- function(x) -1 * (x + 100)

m <- 10
f(m)
```

Self Quiz

```
# Note the name in UseMethod is "g" instead of "f". You should never do this.
f  <- function(x) UseMethod("g")
f.j <- function(x) x + 2
g.j <- function(x) -1 * (x + 2)
f.default <- function(x) x + 100
g.default <- function(x) -1 * (x + 100)
```

```
m <- 10
f(m)
```

```
[1] -110
```