Programación orientada a objetos en lenguaje R

Kevin Zorro 1152312
Nicolás Meneses 1152304
Kevin Tarazona 1152297
David Rincón 1152327
Neffer Rojas 1152307

HISTORIA

R es un lenguaje de programación de código abierto que se originó a partir del lenguaje S, desarrollado por John Chambers y sus colegas en los laboratorios de investigación de Bell en los años 70.

Creadores

Porque usar R y su versión más actualizada

Hay varias razones por las que deberíamos considerar el uso del lenguaje de programación R en el ámbito estadístico y de análisis de datos. Aquí hay algunas de las principales ventajas y diferencias más destacadas de R:

*Amplio conjunto de herramientas y bibliotecas: R cuenta con una amplia colección de paquetes y bibliotecas especializadas en estadística, análisis de datos y visualización. Estos paquetes permiten realizar una amplia gama de tareas, desde análisis descriptivos hasta modelado estadístico avanzado y aprendizaje automático.

*Flexibilidad y capacidad de scripting: R es un lenguaje interpretado y orientado a scripting, lo que permite una gran flexibilidad en la manipulación de datos y el análisis estadístico.

*Enfoque en análisis de datos y estadística: R se ha desarrollado específicamente para el análisis de datos y la estadística, por lo que su conjunto de herramientas y funciones se orienta a estas áreas.

Version actual: 4.2.1 (23 de junio de 2022 (11 meses y 8 días))

Ubicación en el ranking y utilidad del lenguaje R:

R ha sido ampliamente adoptado en la comunidad de análisis de datos y estadística debido a su amplia gama de herramientas y bibliotecas específicas para estas áreas. Según el índice TIOBE, que clasifica los lenguajes de programación según su popularidad, R generalmente ocupa un lugar en el top 20 de los lenguajes más populares. Además, R es ampliamente utilizado en entornos académicos, investigativos y en la industria para tareas relacionadas con el análisis de datos, la visualización, el aprendizaje automático y la estadística.

Concepto de clases y objetos en R

Clase: En el contexto de R, una clase es una estructura que define las propiedades y comportamientos de un tipo de objeto específico

Objeto: Un objeto, por otro lado, es una instancia particular de una clase. Representa una entidad individual que tiene características y puede realizar acciones específicas según lo definido por la clase.

Relación de una clase y sus objetos: La relación entre una clase y sus objetos se basa en el concepto de la instanciación. Una clase define las propiedades y métodos que un objeto puede tener.

Creación de objetos en R

1. Se define una clase

Define la clase utilizando la función setClass(). Especifica el nombre de la clase y las propiedades que deseas que tenga.

```
setClass("MiClase",
         slots = list(propiedad1 = "character", propiedad2 = "numeric"))
```

Creación de objetos en R

2. Se crea el constructor y el objeto

Se crea el constructor utilizando la función setMethod(). El constructor se encarga de crear nuevos objetos y asignarles valores iniciales para las propiedades.

```
setMethod("initialize",  "MiClase”,
        function(propiedad1, propiedad2) {
        new("MiClase", propiedad1 = propiedad1, propiedad2 = propiedad2)
        })
```

Creación de objetos en R

Y a partir de este objeto se puede acceder y manipular las propiedades del mismo utilizando el operador $, seguido del nombre de la propiedad.

```
# Acceder a las propiedades del objeto
print(objeto$propiedad1)  # Imprime: "Hola"
print(objeto$propiedad2)  # Imprime: 10
```

Propiedades de una clase

Las propiedades de una clase en R son los atributos o variables asociados a dicha clase.

Entre estas se pueden hallar:

Variables numéricas: Almacenan valores numéricos, como enteros o números decimales.

```
edad <- 25
salario <- 5000.50
```

Variables de caracteres: Almacenan cadenas de texto.

```
nombre <- "Juan"
apellido <- "Pérez"
```

Propiedades de una clase

Variables lógicas: Almacenan valores de verdadero o falso.

```
esEstudiante <- TRUE
tieneHijos <- FALSE
```

Vectores numéricos: Almacenan una secuencia de valores numéricos.

```
notas <- c(85, 92, 78, 90)
```

Vectores de caracteres: Almacenan una secuencia de cadenas de texto.

```
colores <- c("rojo", "verde", "azul")
```

Propiedades de una clase

Listas: Pueden contener una combinación de diferentes tipos de datos, incluyendo vectores, matrices, data frames y otras listas.

```
informacionPersonal <- list(
nombre = "María",
edad = 30,
notas = c(85, 92, 78, 90),
familia = c("Ana", "Pedro", "Luis")
)
```

Data frames: Almacenan datos tabulares con columnas que pueden contener diferentes tipos de datos.

```
empleados <- data.frame(
nombre = c("Juan", "María", "Pedro"),
edad = c(25, 30, 35),
salario = c(5000, 6000, 7000)
)
```

Condicionales en R

if-else: Es la estructura básica de control condicional en R. Se utiliza para ejecutar un bloque de código si se cumple una condición y otro bloque de código si la condición no se cumple. La sintaxis es la siguiente:

```
if (condición) {
  # Código a ejecutar si la condición es verdadera
} else {
  # Código a ejecutar si la condición es falsa
}
```

if-else if-else: Puedes utilizar esta estructura cuando tienes múltiples condiciones y quieres evaluarlas secuencialmente. El bloque else if se evalúa solo si la condición anterior no se cumple. La sintaxis es la siguiente:

```
if (condición1) {
  # Código a ejecutar si la condición1 es verdadera
} else if (condición2) {
  # Código a ejecutar si la condición2 es verdadera
} else {
  # Código a ejecutar si ninguna de las condiciones anteriores es verdadera
}
```

Condicionales en R

Operador ternario (ifelse): El operador ternario ifelse es una forma concisa de escribir una evaluación condicional en una sola línea. Puedes utilizarlo cuando deseas asignar un valor en función de una condición. La sintaxis es la siguiente:

```
resultado <- ifelse(condición, valor_si_verdadero, valor_si_falso)
```

Ciclos en R

Ciclo for:

El ciclo for se utiliza cuando se conoce de antemano el número de iteraciones que se realizarán, y la variable toma el valor de cada elemento de la secuencia en cada iteración.

```
for (i in 1:5) {
  print(i)
}
```

Ciclo while:

El ciclo while se utiliza cuando la condición para continuar iterando se evalúa en cada iteración, el ciclo se repetirá mientras la condición sea verdadera.

```
i <- 1
while (i <= 5) {
 print(i)
 i <- i + 1
}
```

Ciclos en R

Ciclo repeat y break:

El ciclo repeat se utiliza para ejecutar un bloque de código de forma indefinida hasta que se cumpla una condición específica. El bloque de código dentro del ciclo repeat se ejecuta repetidamente hasta que se encuentre una instrucción break que rompa el ciclo cuando se cumpla una condición.

```
i <- 1
repeat {
 print(i)
 i <- i + 1
 if (i > 5) {
  break
 }
}
```

Métodos de una clase

Los métodos son funciones asociadas a una clase en la programación orientada a objetos. Representan las acciones o comportamientos que los objetos de esa clase pueden realizar.

Definición y llamada de métodos:

Los métodos se definen utilizando las funciones setGeneric() y setMethod(). setGeneric() se utiliza para definir el nombre y la firma del método, y setMethod() se utiliza para asociar el método con una clase y proporcionar su implementación.

Métodos de una clase

Ejemplos:

Constructor: El constructor es un método especial utilizado para crear objetos de una clase. Se utiliza para inicializar las propiedades del objeto. En R, el constructor suele tener el mismo nombre que la clase.

```
setClass("MiClase",
         slots = list(
         propiedad1 = "character",
         propiedad2 = "numeric"
         ))

MiClase <- function(prop1, prop2) {
         new("MiClase", propiedad1 = prop1, propiedad2 = prop2)
         }
```

Métodos de una clase

Ejemplos:

Otros métodos comunes: Además del constructor, se pueden definir otros métodos para realizar diferentes acciones en los objetos de la clase. Por ejemplo, supongamos que tenemos una clase llamada "MiClase" con el método "calcularSuma" que suma las propiedades del objeto.

```
setGeneric("calcularSuma", function(objeto) standardGeneric("calcularSuma"))
setMethod("calcularSuma", "MiClase", function(objeto) {
 suma <- objeto$propiedad1 + objeto$propiedad2
 return(suma)
 })
```

Métodos de una clase

Ejemplos:

Para llamar a un método, utilizamos el nombre del método seguido del objeto al que se aplica y, en algunos casos, los argumentos adicionales que pueda requerir el método.

```
# Llamar al constructor
objeto <- MiClase("valor1", 10)

# Llamar a un método
resultado <- calcularSuma(objeto)
```

Encapsulamiento en R

El encapsulamiento en R se refiere a la práctica de ocultar los detalles internos de una clase u objeto y proporcionar una interfaz controlada para interactuar con ellos.

Algunas de las estrategias para la implementación del encapsulamiento son:

Convención de nomenclatura: Una convención ampliamente utilizada es agregar un prefijo o sufijo al nombre de las propiedades y métodos para indicar su nivel de acceso.

```
MiClase <- setClass("MiClase",
 slots = list(
 _propiedadInterna = "character",
 propiedadPublica = "numeric"
 ),
 prototype = list(
 _propiedadInterna = "valor",
 propiedadPublica = 0
 )
)
```

Encapsulamiento en R

Métodos de acceso: Se pueden definir métodos de acceso para leer o modificar propiedades de una clase. Estos métodos proporcionan una forma controlada de acceder y actualizar los valores de las propiedades.

```
setGeneric("getPropiedadInterna", function(object) standardGeneric("getPropiedadInterna"))
setMethod("getPropiedadInterna", "MiClase", function(object) {
return(object@_propiedadInterna)
})
```

Documentación y comentarios: Es importante proporcionar una documentación clara y comentarios en el código para indicar qué propiedades y métodos son de acceso público y cuáles son de acceso interno. Esto ayuda a los usuarios de la clase a comprender cómo interactuar correctamente con ella.

Contenedores

En el mundo de la programación, los contenedores son como paquetes autónomos que contienen todas las "piezas" necesarias para que una aplicación se ejecute, como el código, las bibliotecas y las configuraciones. Los contenedores permiten empaquetar una aplicación junto con todas sus dependencias en una unidad portátil.
Los contenedores también ofrecen aislamiento, lo que significa que cada contenedor es independiente y no afecta a otros contenedores en ejecución. Esto permite ejecutar varias aplicaciones en el mismo servidor sin que interfieran entre sí.

Asociaciones de clases
Agregación

La agregación es una relación en la que un objeto puede contener o hacer referencia a otros objetos, pero la existencia de los objetos contenidos no depende de la existencia del objeto que los contiene. En otras palabras, los objetos contenidos pueden existir de forma independiente. La agregación gráficamente se suele representar con un rombo claro.

```java
import java.util.ArrayList;
public class Tienda {
    public ArrayList<Cliente> clientes;
    public String nombre;
    public int ventas;

    public Tienda() {
        clientes = new ArrayList<>();
    }

    public Tienda(String nombre, int ventas) {
        this.nombre = nombre;
        this.ventas = ventas;
        clientes = new ArrayList<>();
    }

    public void registrarVenta(Cliente cliente) {
        clientes.add(cliente);
        ventas++;
        cliente.compras++;
    }
}
```

```r
library(R6)
tienda <- R6Class(
  classname = "tienda",
  public = list(
    nombre = NULL,
    clientes = c(),
    ventas = NULL,
    initialize = function(nombre, clientes, ventas) {
      self$nombre <- nombre
      self$clientes <- clientes
      self$ventas <- ventas
    },
    registrarVenta = function(cliente){
      self$clientes <- c(self$clientes, cliente)
      cliente$compras <- cliente$compras + 1
      self$ventas <- self$ventas + 1
    }
  )
)
```

Asociaciones de clases
Composición

La composición es una relación más fuerte en la que un objeto está compuesto por otros objetos y su existencia depende de la existencia del objeto que los contiene. En este caso, los objetos compuestos no pueden existir de forma independiente. La composición gráficamente se suele representar con un rombo oscuro.

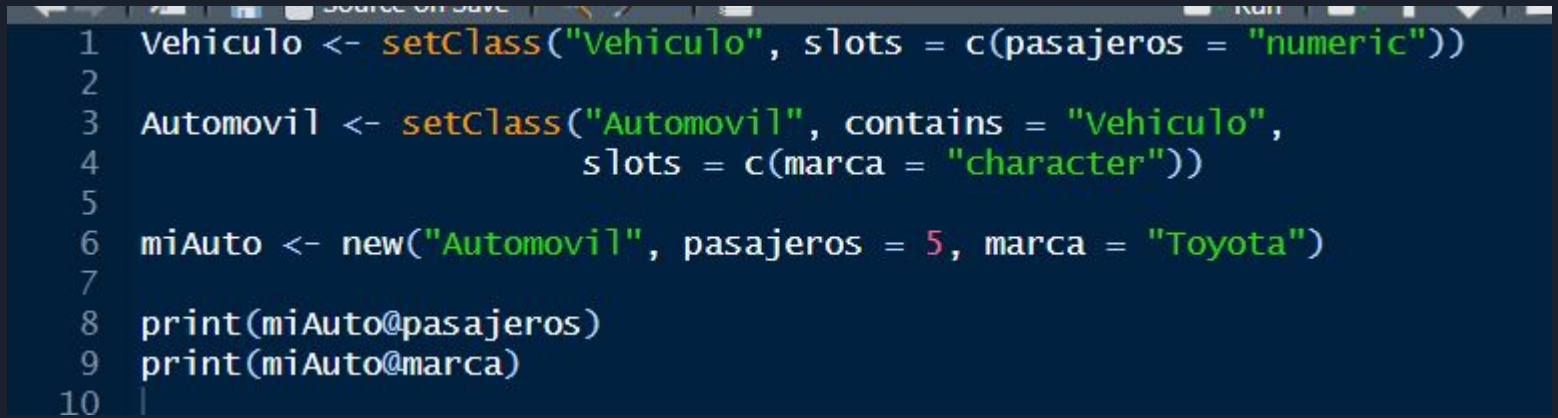
```java
import java.util.ArrayList;
public class Empresa {
    public ArrayList<Empleado> empleados;
    public int numeroEmpleados;
    public String nombre;

    public Empresa() {
        empleados = new ArrayList<>();
    }

    public Empresa(String nombre, int numeroEmpleados) {
        this.nombre = nombre;
        this.numeroEmpleados = numeroEmpleados;
        empleados = new ArrayList<>();
    }

    public void agregarEmpleado(Empleado empleado) {
        empleados.add(empleado);
        numeroEmpleados++;
    }
}
```

```r
library(R6)
empresa <- R6Class(
  classname = "empresa",
  list(
    empleados = c(),
    nombre = NULL,
    numeroEmpleados = NULL,
    initialize = function(nombre, empleados, numeroEmpleados) {
      self$nombre <- nombre
      self$empleados <- empleados
      self$numeroEmpleados <- numeroEmpleados
    },
    agregarEmpleado = function(empleado){
      self$empleados <- c(self$empleados, empleado)
      self$numeroEmpleados <- self$numeroEmpleados + 1
    }
  )
)
```

HERENCIA

La herencia es un concepto fundamental en la programación orientada a objetos (POO) que permite la creación de jerarquías de clases. En la herencia, una clase puede heredar propiedades y comportamientos de otra clase, llamada clase padre o superclase. La clase que hereda se llama clase hija o subclase.

La herencia permite la reutilización de código, ya que las clases hijas heredan automáticamente los campos y métodos de la clase padre. Esto significa que la clase hija puede agregar nuevos campos y métodos o sobrescribir los existentes para adaptarlos a sus necesidades específicas.



```r
Vehiculo <- setClass("Vehiculo", slots = c(pasajeros = "numeric"))

Automovil <- setClass("Automovil", contains = "Vehiculo",
                      slots = c(marca = "character"))

miAuto <- new("Automovil", pasajeros = 5, marca = "Toyota")

print(miAuto@pasajeros)
print(miAuto@marca)
```

Polimorfismo

El polimorfismo es otro concepto clave de la POO que se basa en la idea de que objetos de diferentes clases pueden responder al mismo mensaje o realizar la misma operación de diferentes maneras. El polimorfismo permite tratar objetos de diferentes clases de manera uniforme, siempre que compartan una interfaz común.

En el polimorfismo, una variable puede contener objetos de diferentes clases y, en tiempo de ejecución, se invocará el comportamiento específico de la clase real del objeto. Esto se logra mediante el uso de métodos virtuales o abstractos, que se definen en una clase base y se implementan de manera diferente en las clases hijas.

```
setClass("Figura", slots = c())
setClass("Circulo", contains = "Figura", slots = c(radio = "numeric"))
setClass("Rectangulo", contains = "Figura", slots = c(ancho = "numeric",
                                                      alto = "numeric"))

setGeneric("calcularArea", function(obj) standardGeneric("calcularArea"))
setMethod("calcularArea", "Circulo", function(obj) pi * obj@radio^2)
setMethod("calcularArea", "Rectangulo", function(obj) obj@ancho * obj@alto)

miCirculo <- new("Circulo", radio = 5)
miRectangulo <- new("Rectangulo", ancho = 4, alto = 6)

print(calcularArea(miCirculo))
print(calcularArea(miRectangulo))
```