Introduction Object-oriented Programming

Yang Wu

10/3/2021

Contents

Overview 1
Structure of OOP 2
Main principles of OOP 2
Benefits of OOP 2
OOP in R 3

Overview

The main reason to use OOP is **polymorphism** (literally: many shapes). Polymorphism means that a developer can consider a function's interface separately from its implementation, making it **possible to use the same function form for different types of input.** This is closely related to the idea of **encapsulation**: the user doesn't need to worry about details of an object because they are encapsulated behind a standard interface.

The most important principle of object orientation is **encapsulation**: the idea that data inside the object should only be accessed through a public interface – that is, the object's **methods.** If we want to use the data stored in an object to perform an action or calculate a derived value, *we define a method associated with the object which does this.* Then whenever we want to perform this action we call the method on the object. This is a programming style where implementation details are hidden. It reduces software development complexity greatly. With encapsulation, only methods are exposed. The programmer does not have to worry about implementation details but is only concerned with the operations.

To be more precise, OO systems call the type of an object its **class**, and an implementation for a specific class is called a **method**. Roughly speaking, *a class defines what an object is and methods describe what that object can do.* The class defines the fields, the data possessed by every instance of that class. Classes are organised in a hierarchy so that if a method does not exist for one class, its parent's method is used, and the child is said to **inherit** behavior. Inheritance allows you to create class hierarchies, where a base class gives its behavior and attributes to a derived class. You are then free to modify or extend its functionality. Polymorphism ensures that the proper method will be executed based on the calling object's type. The process of finding the correct method given a class is called **method dispatch**.

- In **encapsulated OOP**, methods belong to objects or classes, and method calls typically look like `object.method(arg1, arg2)`. This is called encapsulated because the object encapsulates both data (with fields) and behavior (with methods), and is the paradigm found in most popular languages.
- In **functional (or generic function) OOP**, methods belong to generic functions, and method calls look like ordinary function calls: `generic(object, arg2, arg3)`. This is called functional because from the outside it looks like a regular function call, and internally the components are also functions.

Structure of OOP

- **Classes** are user-defined data types that act as the blueprint for individual objects, attributes and methods.
- **Objects** are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.
- **Methods** are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for re-usability or keeping functionality encapsulated inside one object at a time.
- **Attributes** are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.

Main principles of OOP

- **Encapsulation.** This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.
- **Abstraction.** Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.
- **Inheritance.** Classes can reuse code from other classes. Relationships and sub-classes between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.
- **Polymorphism.** Objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.

Benefits of OOP

- **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
- **Re-usability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Productivity.** Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
- **Easily upgradable and scalable.** Programmers can implement system functionalities independently.

- **Interface descriptions.** Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
- **Security.** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
- **Flexibility.** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

OOP in R

Base R provides three OOP systems: S3, S4, and reference classes (RC):

- S3 is R's first OOP system, and is described in Statistical Models in S.62 S3 is an informal implementation of functional OOP and relies on common conventions rather than ironclad guarantees. This makes it easy to get started with, providing a low cost way of solving many simple problems.
- S4 is a formal and rigorous rewrite of S3, and was introduced in Programming with Data.63 It requires more upfront work than S3, but in return provides more guarantees and greater encapsulation. S4 is implemented in the base methods package, which is always installed with R.

(You might wonder if S1 and S2 exist. They don't: S3 and S4 were named according to the versions of S that they accompanied. The first two versions of S didn't have any OOP framework.)

- RC implements encapsulated OO. RC objects are a special type of S4 objects that are also mutable, i.e., instead of using R's usual copy-on-modify semantics, they can be modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve in the functional OOP style of S3 and S4.

A number of other OOP systems are provided by CRAN packages:

- R6 implements encapsulated OOP like RC, but resolves some important issues.
- R.oo provides some formalism on top of S3, and makes it possible to have mutable S3 objects.
- proto implements another style of OOP based on the idea of prototypes, which blur the distinctions between classes and instances of classes (objects).