WIKIPEDIA

Encapsulation (computer programming)

In object-oriented programming (OOP), **encapsulation** refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.[1] Encapsulation is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties' direct access to them. Publicly accessible methods are generally provided in the class (so-called "getters" and "setters") to access the values, and other client classes call these methods to retrieve and modify the values within the object.

This mechanism is not unique to OOP. Implementations of abstract data types, e.g., modules, offer a similar form of encapsulation. The similarity has been explained by programming language theorists in terms of existential types.[2]

Contents

Meaning

In object-oriented programming languages, and other related fields, encapsulation refers to one of two related but distinct notions, and sometimes to the combination thereof:[3][4]

- A language mechanism for restricting direct access to some of the object's components.[5][6]
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.[1][7]

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object-oriented programming, while some programming languages that provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many object-oriented languages, and other related fields, the components are not hidden automatically and this can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.

The features of encapsulation are supported using classes in most object-oriented languages, although other alternatives also exist.

Encapsulation and inheritance

The authors of *Design Patterns* discuss the tension between inheritance and encapsulation at length and state that in their experience, designers overuse inheritance. They claim that inheritance often breaks encapsulation, given that inheritance exposes a subclass to the details of its parent's implementation.[8] As described by the Yo-yo problem, overuse of inheritance and therefore encapsulation, can become too complicated and hard to debug.

Information hiding

Under the definition that encapsulation "can be used to hide data members and member functions", the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields. Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A supposed benefit of encapsulation is that it can reduce system complexity, and thus increase robustness, by allowing the developer to limit the interdependencies between software components.

Some languages like Smalltalk and Ruby only allow access via object methods, but most others (e.g., C++, C#, Delphi or Java) offer the programmer a degree of control over what is hidden, typically via keywords like `public` and `private`.[6] ISO C++ standard refers to `protected`, `private` and `public` as "access specifiers" and that they do not "hide any information". Information hiding is accomplished by furnishing a compiled version of the source code that is interfaced via a header file.

Almost always, there is a way to override such protection – usually via reflection API (Ruby, Java, C#, etc.), sometimes by mechanism like name mangling (Python), or special keyword usage like `friend` in C++.

Examples

Restricting data fields

Languages like C++, C#, Java, PHP, Swift, and Delphi offer ways to restrict access to data fields.

Below is an example in C# that shows how access to a data field can be restricted through the use of a `private` keyword:

```
class Program {
    public class Account {
        private decimal accountBalance = 500.00m;

        public decimal CheckBalance() {
            return accountBalance;
        }
    }

    static void Main() {
        Account myAccount = new Account();
        decimal myBalance = myAccount.CheckBalance();

        /* This Main method can check the balance via the public
        * "CheckBalance" method provided by the "Account" class
        * but it cannot manipulate the value of "accountBalance" */
    }
}
```

Below is an example in Java:

```
public class Employee {
    private BigDecimal salary = new BigDecimal(50000.00);

    public BigDecimal getSalary() {
        return salary;
    }

    public static void main() {
        Employee e = new Employee();
        BigDecimal sal = e.getSalary();
    }
}
```

Encapsulation is also possible in non-object-oriented languages. In C, for example, a structure can be declared in the public API via the header file for a set of functions that operate on an item of data containing data members that are not accessible to clients of the API with the `extern` keyword.[9][10]

```
// Header file "api.h"

struct Entity;          // Opaque structure with hidden members

// API functions that operate on 'Entity' objects
extern struct Entity *  open_entity(int id);
```

```
extern int              process_entity(struct Entity *info);
extern void             close_entity(struct Entity *info);
// extern keywords here are redundant, but don't hurt.
// extern defines functions that can be called outside the current file, the default behavior even without the keyword
```

Clients call the API functions to allocate, operate on, and deallocate objects of an opaque data type. The contents of this type are known and accessible only to the implementation of the API functions; clients cannot directly access its contents. The source code for these functions defines the actual contents of the structure:

```
// Implementation file "api.c"

#include "api.h"

struct Entity {
    int     ent_id;         // ID number
    char    ent_name[20];   // Name
    ... and other members ...
};

// API function implementations
struct Entity * open_entity(int id)
{ ... }

int process_entity(struct Entity *info)
{ ... }

void close_entity(struct Entity *info)
{ ... }
```

Name mangling

Below is an example of Python, which does not support variable access restrictions. However, the convention is that a variable whose name is prefixed by an underscore should be considered private.[11]

```
class Car(object):
    def __init__(self) -> None:
        self._maxspeed = 200

    def drive(self) -> None:
        print(f'Maximum speed is {self._maxspeed}.')

redcar = Car()
redcar.drive()  # This will print 'Maximum speed is 200.'

redcar._maxspeed = 10
redcar.drive()  # This will print 'Maximum speed is 10.'
```

See also

- Inheritance (object-oriented programming)
- Object-oriented programming
- Software design pattern
- Facade pattern

References

1. Rogers, Wm. Paul (18 May 2001). "Encapsulation is not information hiding" (http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html?page=9). JavaWorld.
2. Pierce 2002, § 24.2 Data Abstraction with Existentials
3. Scott, Michael Lee (2006). *Programming language pragmatics* (2 ed.). Morgan Kaufmann. p. 481. ISBN 978-0-12-633951-2. "Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction."
4. Dale, Nell B.; Weems, Chip (2007). *Programming and problem solving with Java* (2nd ed.). Jones & Bartlett. p. 396. ISBN 978-0-7637-3402-2.
5. Mitchell, John C. (2003). *Concepts in programming languages*. Cambridge University Press. p. 522. ISBN 978-0-521-78098-8.
6. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. p. 266. ISBN 978-0-262-16209-8.
7. Connolly, Thomas M.; Begg, Carolyn E. (2005). "Ch. 25: Introduction to Object DMBS § Object-oriented concepts". *Database systems: a practical approach to design, implementation, and management* (4th ed.). Pearson Education. p. 814. ISBN 978-0-321-21025-8.
8. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns* (https://archive.org/details/designpatternsel00gamm). Addison-Wesley. ISBN 978-0-201-63361-0.
9. King, K. N. (2008). *C Programming: A Modern Approach* (https://www.stormingrobots.com/prod/tutorial/pdf/kingBook-ch1to10.pdf) (PDF) (2nd ed.). W. W. Norton & Company. p. 464. ISBN 978-0393979503. Retrieved 1 November 2019.
10. King, Kim N. *C programming: a modern approach*. WW Norton & Company, 2008. Ch. 18, p. 464, ISBN 0393979504
11. Bader, Dan. "The Meaning of Underscores in Python" (https://dbader.org/blog/meaning-of-underscores-in-python). *Improve Your Python Skills*. Retrieved 1 November 2019.