WIKIPEDIA

Encapsulation (computer programming)

In object oriented programming languages, **encapsulation** is used to refer to one of two related but distinct notions, and sometimes to the combination[1][2] thereof:

- A language mechanism for restricting direct access to some of the object's components.[3][4]
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.[5][6]

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 of the OOP languages hiding of components is not automatic or 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 programming languages, although other alternatives also exist.

Contents

General definition

Encapsulation *is one of the fundamentals* of OOP (object-oriented programming). It refers to the bundling of data with the methods that operate on that data.[5] 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 object-oriented programming. Implementations of abstract data types, e.g. modules, offer a similar form of encapsulation. This similarity stems from the fact that both notions rely on the same mathematical fundamental of an existential type.[7]

An information-hiding mechanism

Encapsulation can be used to hide data members and member functions. Under this definition, encapsulation means that 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. 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`.[4] 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.

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 inter-dependencies between software components.

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++.

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();
    }
}
```

Below is an example in PHP:

```
class Account {
    /**
     * How much money is currently in the account
     * @var float
     */
    private $accountBalance;

    /**
     * @param float $currentAccountBalance Initialize account to this dollar amount
     */
    public function __construct($currentAccountBalance) {
        $this->accountBalance = $currentAccountBalance;
    }

    /**
     * Add money to account
     * @param float $money Dollars to add to balance
     * @return void
     */
    public function deposit($money) {
        $this->accountBalance += $money;
    }

    /**
     * Remove money from account
     * @param float $money Dollars to subtract from balance
     * @throws Exception
     * @return void
     */
    public function withdraw($money) {
        if ($this->accountBalance < $money) {
            throw new Exception('Cannot withdraw $' . $money . ' from account as it contains $' . $this->accountBalance);
        }
        $this->accountBalance -= $money;
    }

    /**
     * Get current account balance, that takes all additions and subtractions into consideration.
     * @return float
     */
    public function getAccountBalance() {
        return $this->accountBalance;
    }
}

// Create a new object from the Account class with a starting balance of $500.00
$myAccount = new Account(500.00);

// We have clearly defined methods for adding and subtracting money from the Account
// If we didn't have a method for withdraw(), nothing would prevent us from withdrawing more money than was available in the account
$myAccount->deposit(10.24);
$myAccount->withdraw(4.45);

// Get the current balance
$accountBalance = $myAccount->getAccountBalance();
echo 'My Account Balance: $' . $accountBalance; // 505.79

// Our code forbids us from withdrawing more than we have
$myAccount->withdraw(600.00); // Exception Message: Cannot withdraw $600 from account as it contains $505.79
```

Encapsulation is also possible in non-object-oriented languages. In C, for example, a structure can be declared in the public API (i.e., 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:

```
// 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 they won't hurt.
// extern keyword defines functions to have external linkage (i.e. can be called outside current file)
// which is the default behavior even without the keyword extern
```

Note on extern keyword from K.N. King.[8]

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"

// Complete definition of the 'Entity' object
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)
{ ... }
```

Encapsulation and inheritance

The authors of Design Patterns[9] discuss the tension between inheritance and encapsulation at length and state that in their experience, designers overuse inheritance. The danger is stated as follows:

> Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation"
>
> — Gang of Four, Design Patterns[9] (Chapter 1)

References

1. 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."
2. Dale, Nell B.; Weems, Chip (2007). *Programming and problem solving with Java* (2nd ed.). Jones & Bartlett. p. 396. ISBN 978-0-7637-3402-2.
3. Mitchell, John C. (2003). *Concepts in programming languages*. Cambridge University Press. p. 522. ISBN 978-0-521-78098-8.
4. Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. p. 266. ISBN 978-0-262-16209-8.
5. 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.
6. 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.
7. Pierce 2002, § 24.2 Data Abstraction with Existentials
8. King, Kim N. C programming: a modern approach. WW Norton & Company, 2008. Ch. 18, p. 464, ISBN 0393979504
9. Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns*. Addison-Wesley. ISBN 978-0-201-63361-0.

External links

- Object-Oriented Encapsulation Definition (http://wiki.c2.com/?EncapsulationDefinition)

- SOA Patterns.org (http://www.soapatterns.org/service_encapsulation.php)
- Encapsulation (http://telecomacadmey.com/understanding-encapsulation-troubleshooting-network/)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Encapsulation_(computer_programming)&oldid=889772980"

This page was last edited on 27 March 2019, at 21:16 (UTC).