Class (computer programming)

In object-oriented programming, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).[1][2] In many languages, the class name is used as the name for the class (the template itself), the name for the default constructor of the class (a subroutine that creates objects), and as the type of objects generated by instantiating the class; these distinct concepts are easily conflated.[2]

When an object is created by a constructor of the class, the resulting object is called an instance of the class, and the member variables specific to the object are called instance variables, to contrast with the class variables shared across the class.

In some languages, classes are only a compile-time feature (new classes cannot be declared at runtime), while in other languages classes are first-class citizens, and are generally themselves objects (typically of type `Class` or similar). In these languages, a class that creates classes is called a metaclass.

Contents

Class vs. type

In casual use, people often refer to the "class" of an object, but narrowly speaking objects have *type*: the interface, namely the types of member variables, the signatures of member functions (methods), and properties these satisfy. At the same time, a class has an implementation (specifically the implementation of the methods), and can create objects of a given type, with a given implementation.[3] In the terms of type theory, a class is an implementation—a *concrete* data structure and collection of subroutines—while a type is an interface. Different (concrete) classes can produce objects of the same (abstract) type (depending on type system); for example, the type `Stack` might be implemented with two classes – `SmallStack` (fast for small stacks, but scales poorly) and `ScalableStack` (scales well but high overhead for small stacks). Similarly, a given class may have several different constructors.

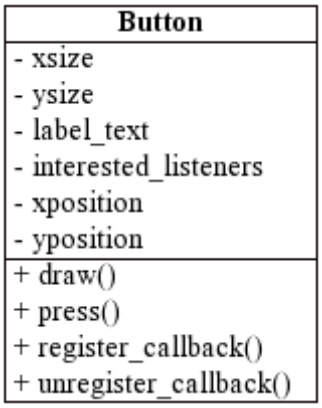
Types generally represent nouns, such as a person, place or thing, or something nominalized, and a class represents an implementation of these. For example, a `Banana` type might represent the properties and functionality of bananas in general, while the `ABCBanana` and `XYZBanana` classes would represent ways of producing bananas (say, banana suppliers or data structures and functions to represent and draw bananas in a video game). The `ABCBanana` class could then produce particular bananas: instances of the `ABCBanana` class would be objects of type `Banana`. Often only a single implementation of a type is given, in which case the class name is often identical with the type name.

Design and implementation

Classes are composed from structural and behavioral constituents.[4] Programming languages that include classes as a programming construct offer support, for various class-related features, and the syntax required to use these features varies greatly from one programming language to another.

Structure

A class contains data field descriptions (or *properties*, *fields*, *data members*, or *attributes*). These are usually field types and names that will be associated with state variables at program run time; these state variables either belong to the class or specific instances of the class. In most languages, the structure defined by the class determines the layout of the memory used by its instances. Other implementations are possible: for example, objects in Python use associative key-value containers.[5]



UML notation for classes

Some programming languages support specification of invariants as part of the definition of the class, and enforce them through the type system. Encapsulation of state is necessary for being able to enforce the invariants of the class.

Behavior

The behavior of class or its instances is defined using methods. Methods are subroutines with the ability to operate on objects or classes. These operations may alter the state of an object or simply provide ways of accessing it.[6] Many kinds of methods exist, but support for them varies across languages. Some types of methods are created and called by programmer code, while other special methods—such as constructors, destructors, and conversion operators—are created and called by compiler-generated code. A language may also allow the programmer to define and call these special methods.[7][8]

The concept of class interface

Every class *implements* (or *realizes*) an interface by providing structure and behavior. Structure consists of data and state, and behavior consists of code that specifies how methods are implemented.[9] There is a distinction between the definition of an interface and the implementation of that interface; however, this line is blurred in many programming languages because class declarations both define and implement an interface. Some languages, however, provide features that separate interface and implementation. For example, an abstract class can define an interface without providing implementation.

Languages that support class inheritance also allow classes to inherit interfaces from the classes that they are derived from.

For example,if "class A" inherits from "class B" and if "class B" implements the interface "interface B" then "class A" also inherits the functionality(constants and methods declaration) provided by "interface B".

In languages that support access specifiers, the interface of a class is considered to be the set of public members of the class, including both methods and attributes (via implicit getter and setter methods); any private members or internal data structures are not intended to be depended on by external code and thus are not part of the interface.

Object-oriented programming methodology dictates that the operations of any interface of a class are to be independent of each other. It results in a layered design where clients of an interface use the methods declared in the interface. An interface places no requirements for clients to invoke the operations of one interface in any particular order. This approach has the benefit that client code can assume that the operations of an interface are available for use whenever the client has access to the object.

Example

The buttons on the front of your television set are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to toggle the television on and off. In this example, your particular television is the instance, each method is represented by a button, and all the buttons together comprise the interface. (Other television sets that are the same model as yours would have the same interface.) In its most common form, an interface is a specification of a group of related methods without any associated implementation of the methods.

A television set also has a myriad of *attributes*, such as size and whether it supports color, which together comprise its structure. A class represents the full description of a television, including its attributes (structure) and buttons (interface).

Getting the total number of televisions manufactured could be a *static method* of the television class. This method is clearly associated with the class, yet is outside the domain of each individual instance of the class. Another example would be a static method that finds a particular instance out of the set of all television objects.

Member accessibility

The following is a common set of access specifiers:[10]

- *Private* (or *class-private*) restricts the access to the class itself. Only methods that are part of the same class can access private members.
- *Protected* (or *class-protected*) allows the class itself and all its subclasses to access the member.
- *Public* means that any code can access the member by its name.

Although many object-oriented languages support the above access specifiers, their semantics may differ.

Object-oriented design uses the access specifiers in conjunction with careful design of public method implementations to enforce class invariants—constraints on the state of the objects. A common usage of access specifiers is to separate the internal data of a class from its interface: the internal structure is made private, while public accessor methods can be used to inspect or alter such private data.

Access specifiers do not necessarily control *visibility*, in that even private members may be visible to client external code. In some languages, an inaccessible but visible member may be referred to at run-time (for example, by a pointer returned from a member function), but an attempt to use it by referring to the name of the member from client code will be prevented by the type checker.[11]

The various object-oriented programming languages enforce member accessibility and visibility to various degrees, and depending on the language's type system and compilation policies, enforced at either compile-time or run-time. For example, the Java language does not allow client code that accesses the private data of a class to compile. [12] In the C++ language, private methods are visible, but not accessible in the interface; however, they may be made invisible by explicitly declaring fully abstract classes that represent the interfaces of the class.[13]

Some languages feature other accessibility schemes:

- *Instance vs. class accessibility*: Ruby supports *instance-private* and *instance-protected* access specifiers in lieu of class-private and class-protected, respectively. They differ in that they restrict access based on the instance itself, rather than the instance's class.[14]
- *Friend*: C++ supports a mechanism where a function explicitly declared as a friend function of the class may access the members designated as private or protected.[15]
- *Path-based*: Java supports restricting access to a member within a Java package, which is the logical path of the file. However, it is a common practice when extending a Java framework to implement classes in the same package as a

framework class in order to access protected members. The source file may exist in a completely different location, and may be deployed to a different .jar file, yet still be in the same logical path as far as the JVM is concerned.[10]

Inter-class relationships

In addition to the design of standalone classes, programming languages may support more advanced class design based upon relationships between classes. The inter-class relationship design capabilities commonly provided are *compositional* and *hierarchical*.

Compositional

Classes can be composed of other classes, thereby establishing a compositional relationship between the enclosing class and its embedded classes. Compositional relationship between classes is also commonly known as a *has-a* relationship.[16] For example, a class "Car" could be composed of and contain a class "Engine". Therefore, a Car *has an* Engine. One aspect of composition is containment, which is the enclosure of component instances by the instance that has them. If an enclosing object contains component instances by value, the components and their enclosing object have a similar lifetime. If the components are contained by reference, they may not have a similar lifetime.[17] For example, in Objective-C 2.0:

```
@interface Car : NSObject

@property NSString *name;
@property Engine *engine
@property NSArray *tires;

@end
```

This `Car` class *has* an instance of `NSString` (a string object), `Engine`, and `NSArray` (an array object).

Hierarchical

Classes can be *derived* from one or more existing classes, thereby establishing a hierarchical relationship between the derived-from classes (*base classes*, *parent classes* or *superclasses*) and the derived class (*child class* or *subclass*) . The relationship of the derived class to the derived-from classes is commonly known as an **is-a** relationship.[18] For example, a class 'Button' could be derived from a class 'Control'. Therefore, a Button **is a** Control. Structural and behavioral members of the parent classes are *inherited* by the child class. Derived classes can define additional structural members (data fields) and behavioral members (methods) in addition to those that they *inherit* and are therefore *specializations* of their superclasses. Also, derived classes can override inherited methods if the language allows.

Not all languages support multiple inheritance. For example, Java allows a class to implement multiple interfaces, but only inherit from one class.[19] If multiple inheritance is allowed, the hierarchy is a directed acyclic graph (or DAG for short), otherwise it is a tree. The hierarchy has classes as nodes and inheritance relationships as links. Classes in the same level are more likely to be associated than classes in different levels. The levels of this hierarchy are called layers or levels of abstraction.

Example (Simplified Objective-C 2.0 code, from iPhone SDK):

```
@interface UIResponder : NSObject //...
@interface UIView : UIResponder //...
@interface UIScrollView : UIView //...
@interface UITableView : UIScrollView //...
```

In this example, a UITableView **is a** UIScrollView **is a** UIView **is a** UIResponder **is an** NSObject.

Definitions of subclass

Conceptually, a superclass is a superset of its subclasses. For example, a common class hierarchy would involve `GraphicObject` as a superclass of `Rectangle` and `Elipse`, while `Square` would be a subclass of `Rectangle`. These are all subset relations in set theory as well, i.e., all squares are rectangles but not all rectangles are squares.

A common conceptual error is to mistake a *part of* relation with a subclass. For example, a car and truck are both kinds of vehicles and it would be appropriate to model them as subclasses of a vehicle class. However, it would be an error to model the component parts of the car as subclass relations. For example, a car is composed of an engine and body, but it would not be appropriate to model engine or body as a subclass of car.

In object-oriented modeling these kinds of relations are typically modeled as object properties. In this example the Car class would have a property called `parts`. `parts` would be typed to hold a collection of objects such as instances of `Body, Engine, Tires,...`. Object modeling languages such as UML include capabilities to model various aspects of part of and other kinds of relations. Data such as the cardinality of the objects, constraints on input and output values, etc. This information can be utilized by developer tools to generate additional code beside the basic data definitions for the objects. Things such as error checking on get and set methods.[20]

One important question when modeling and implementing a system of object classes is whether a class can have one or more superclasses. In the real world with actual sets it would be rare to find sets that didn't intersect with more than one other set. However, while some systems such as Flavors and CLOS provide a capability for more than one parent to do so at run time introduces complexity that many in the object-oriented community consider antithetical to the goals of using object classes in the first place. Understanding which class will be responsible for handling a message can get complex when dealing with more than one superclass. If used carelessly this feature can introduce some of the same system complexity and ambiguity classes were designed to avoid.[21]

Most modern object-oriented languages such as Smalltalk and Java require single inheritance at run time. For these languages, multiple inheritance may be useful for modeling but not for an implementation.

However, semantic web application objects do have multiple superclasses. The volatility of the Internet requires this level of flexibility and the technology standards such as the Web Ontology Language (OWL) are designed to support it.

A similar issue is whether or not the class hierarchy can be modified at run time. Languages such as Flavors, CLOS, and Smalltalk all support this feature as part of their meta-object protocols. Since classes are themselves first-class objects, it is possible to have them dynamically alter their structure by sending them the appropriate messages. Other languages that focus more on strong typing such as Java and C++ do not allow the class hierarchy to be modified at run time. Semantic web objects have the capability for run time changes to classes. The rational is similar to the justification for allowing multiple superclasses, that the Internet is so dynamic and flexible that dynamic changes to the hierarchy are required to manage this volatility.[22]

Orthogonality of the class concept and inheritance

Although class-based languages are commonly assumed to support inheritance, inheritance is not an intrinsic aspect of the concept of classes. Some languages, often referred to as "object-based languages", support classes yet do not support inheritance. Examples of object-based languages include earlier versions of Visual Basic.

Within object-oriented analysis

In object-oriented analysis and in UML, an **association** between two classes represents a collaboration between the classes or their corresponding instances. Associations have direction; for example, a bi-directional association between two classes indicates that both of the classes are aware of their relationship.[23] Associations may be labeled according to their name or purpose.[24]

An association role is given end of an association and describes the role of the corresponding class. For example, a "subscriber" role describes the way instances of the class "Person" participate in a "subscribes-to" association with the class "Magazine". Also, a "Magazine" has the "subscribed magazine" role in the same association. Association role multiplicity describes how many instances correspond to each instance of the other class of the association. Common multiplicities are "0..1", "1..1", "1..*" and "0..*", where the "*" specifies any number of instances.[23]

Taxonomy of classes

There are many categories of classes, some of which overlap.

Abstract and concrete

In a language that supports inheritance, an **abstract class**, or *abstract base class* (ABC), is a class that cannot be instantiated because it is either labeled as abstract or it simply specifies abstract methods (or *virtual methods*). An abstract class may provide implementations of some methods, and may also specify virtual methods via signatures that are to be implemented by direct or indirect descendants of the abstract class. Before a class derived from an abstract class can be instantiated, all abstract methods of its parent classes must be implemented by some class in the derivation chain.[25]

Most object-oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java, C# and PHP, the keyword *abstract* is used.[26][27] In C++, an abstract class is a class having at least one abstract method given by the appropriate syntax in that language (a pure virtual function in C++ parlance).[25]

A class consisting of only virtual methods is called a Pure Abstract Base Class (or *Pure ABC*) in C++ and is also known as an *interface* by users of the language.[13] Other languages, notably Java and C#, support a variant of abstract classes called an interface via a keyword in the language. In these languages, multiple inheritance is not allowed, but a class can implement multiple interfaces. Such a class can only contain abstract publicly accessible methods.[19][28][29]

A **concrete class** is a class that can be instantiated, as opposed to abstract classes, which cannot.

Local and inner

In some languages, classes can be declared in scopes other than the global scope. There are various types of such classes.

An **inner class** is a class defined within another class. The relationship between an inner class and its containing class can also be treated as another type of class association. An inner class is typically neither associated with instances of the enclosing class nor instantiated along with its enclosing class. Depending on language, it may or may not be possible to refer to the class from outside the enclosing class. A related concept is *inner types*, also known as *inner data type* or *nested type*, which is a generalization of the concept of inner classes. C++ is an example of a language that supports both inner classes and inner types (via *typedef* declarations).[30][31]

Another type is a **local class**, which is a class defined within a procedure or function. This limits references to the class name to within the scope where the class is declared. Depending on the semantic rules of the language, there may be additional restrictions on local classes compared to non-local ones. One common restriction is to disallow local class methods to access local variables of the enclosing function. For example, in C++, a local class may refer to static variables declared within its enclosing function, but may not access the function's automatic variables.[32]

Metaclasses

Metaclasses are classes whose instances are classes.[33] A metaclass describes a common structure of a collection of classes and can implement a design pattern or describe particular kinds of classes. Metaclasses are often used to describe frameworks.[34]

In some languages, such as Python, Ruby or Smalltalk, a class is also an object; thus each class is an instance of a unique metaclass that is built into the language. [5] [35] [36] The Common Lisp Object System (CLOS) provides metaobject protocols (MOPs) to implement those classes and metaclasses. [37]

Non-subclassable

Non-subclassable classes allow programmers to design classes and hierarchies of classes where at some level in the hierarchy, further derivation is prohibited. (A stand-alone class may be also designated as non-subclassable, preventing the formation of any hierarchy). Contrast this to *abstract* classes, which imply, encourage, and require derivation in order to be used at all. A non-subclassable class is implicitly *concrete*.

A non-subclassable class is created by declaring the class as `sealed` in C# or as `final` in Java or PHP.[38][39][40]

For example, Java's `String` class is designated as *final*.[41]

Non-subclassable classes may allow a compiler (in compiled languages) to perform optimizations that are not available for subclassable classes.

Open Class

An open class is one that can be changed. Typically, an executable program cannot be changed by customers. Developers can often change some classes, but typically cannot change standard or built-in ones. In Ruby, all classes are open. In Python, classes can be created at runtime, and all can be modified afterwards.[42] Objective-C categories permit the programmer to add methods to an existing class without the need to recompile that class or even have access to its source code.

Mixins

Some languages have special support for mixins, though in any language with multiple inheritance a mixin is simply a class that does not represent an is-a-type-of relationship. Mixins are typically used to add the same methods to multiple classes; for example, a class `UnicodeConversionMixin` might provide a method called `unicode_to_ascii` when included in classes `FileReader` and `WebPageScraper` that do not share a common parent.

Partial

In languages supporting the feature, a **partial class** is a class whose definition may be split into multiple pieces, within a single source-code file or across multiple files.[43] The pieces are merged at compile-time, making compiler output the same as for a non-partial class.

The primary motivation for introduction of partial classes is to facilitate the implementation of code generators, such as visual designers.[43] It is otherwise a challenge or compromise to develop code generators that can manage the generated code when it is interleaved within developer-written code. Using partial classes, a code generator can process a separate file or coarse-grained partial class within a file, and is thus alleviated from intricately interjecting generated code via extensive parsing, increasing compiler efficiency and eliminating the potential risk of corrupting developer code. In a simple implementation of partial classes, the compiler can perform a phase of precompilation where it "unifies" all the parts of a partial class. Then, compilation can proceed as usual.

Other benefits and effects of the partial class feature include:

- Enables separation of a class's interface and implementation code in a unique way.
- Eases navigation through large classes within an editor.
- Enables separation of concerns, in a way similar to aspect-oriented programming but without using any extra tools.
- Enables multiple developers to work on a single class concurrently without the need to merge individual code into one file at a later time.

Partial classes have existed in Smalltalk under the name of *Class Extensions* for considerable time. With the arrival of the .NET framework 2, Microsoft introduced partial classes, supported in both C# 2.0 and Visual Basic 2005. WinRT also supports partial classes.

Example in VB.NET

This simple example, written in Visual Basic .NET, shows how parts of the same class are defined in two different files.

file1.vb

```
Partial Class MyClass
    Private _name As String
End Class
```

file2.vb

```
Partial Class MyClass
    Public Readonly Property Name() As String
        Get
```

```
            Return _name
        End Get
    End Property
End Class
```

When compiled, the result is the same as if the two files were written as one, like this:

```
Class MyClass
    Private _name As String
    Public Readonly Property Name() As String
        Get
            Return _name
        End Get
    End Property
End Class
```

Example in Objective-C

In Objective-C, partial classes, also known as **categories**, may even spread over multiple libraries and executables, like the following example. But a key difference is that a Objective-C's categories can overwrite definitions in another interface declaration, and that categories aren't equal to original class definition (the first requires the last).[44] Instead, .NET partial class can't have conflicting definitions, and all partial definitions are equal to the others.[43]

In Foundation, header file NSData.h:

```
@interface NSData : NSObject

- (id)initWithContentsOfURL:(NSURL *)URL;
//...

@end
```

In user-supplied library, a separate binary from Foundation framework, header file NSData+base64.h:

```
#import <Foundation/Foundation.h>

@interface NSData (base64)

- (NSString *)base64String;
- (id)initWithBase64String:(NSString *)base64String;

@end
```

And in an app, yet another separate binary file, source code file main.m:

```
#import <Foundation/Foundation.h>
#import "NSData+base64.h"

int main(int argc, char *argv[])
{
    if (argc < 2)
        return EXIT_FAILURE;
    NSString *sourceURLString = [NSString stringWithCString:argv[1]];
    NSData *data = [[NSData alloc] initWithContentsOfURL:[NSURL URLWithString:sourceURLString]];
    NSLog(@"%@", [data base64String]);
    return EXIT_SUCCESS;
}
```

The dispatcher will find both methods called over the NSData instance and invoke both of them correctly.

Uninstantiable

Uninstantiable classes allow programmers to group together per-class fields and methods that are accessible at runtime without an instance of the class. Indeed, instantiation is prohibited for this kind of class.

For example, in C#, a class marked "static" can not be instantiated, can only have static members (fields, methods, other), may not have *instance constructors*, and is *sealed*. [45]

Unnamed

An **unnamed class** or **anonymous class** is a class that is not bound to a name or identifier upon definition. This is analogous to named versus unnamed functions.

Benefits

The benefits of organizing software into object classes fall into three categories:[46]

- Rapid development
- Ease of maintenance
- Reuse of code and designs

Object classes facilitate rapid development because they lessen the semantic gap between the code and the users. System analysts can talk to both developers and users using essentially the same vocabulary, talking about accounts, customers, bills, etc. Object classes often facilitate rapid development because most object-oriented environments come with powerful debugging and testing tools. Instances of classes can be inspected at run time to verify that the system is performing as expected. Also, rather than get dumps of core memory, most object-oriented environments have interpreted debugging capabilities so that the developer can analyze exactly where in the program the error occurred and can see which methods were called to which arguments and with what arguments.[47]

Object classes facilitate ease of maintenance via encapsulation. When developers need to change the behavior of an object they can localize the change to just that object and its component parts. This reduces the potential for unwanted side effects from maintenance enhancements.

Software re-use is also a major benefit of using Object classes. Classes facilitate re-use via inheritance and interfaces. When a new behavior is required it can often be achieved by creating a new class and having that class inherit the default behaviors and data of its superclass and then tailor some aspect of the behavior or data accordingly. Re-use via interfaces (also known as methods) occurs when another object wants to invoke (rather than create a new kind of) some object class. This method for re-use removes many of the common errors that can make their way into software when one program re-uses code from another.[48]

Run-time representation

As a data type, a class is usually considered as a compile-time construct. A language may also support prototype or factory metaobjects that represent run-time information about classes, or even represent metadata that provides access to reflection facilities and ability to manipulate data structure formats at run-time. Many languages distinguish this kind of run-time type information about classes from a class on the basis that the information is not needed at run-time. Some dynamic languages do not make strict distinctions between run-time and compile-time constructs, and therefore may not distinguish between metaobjects and classes.

For example, if Human is a metaobject representing the class Person, then instances of class Person can be created by using the facilities of the Human metaobject.

See also

- Class-based programming
- Class diagram (UML)
- List of object-oriented programming languages
- Mixin
- Object-oriented programming
- Prototype-based programming
- Trait (computer programming)

Notes

1. Gamma et al. 1995, p. 14.
2. Bruce 2002, 2.1 Objects, classes, and object types, https://books.google.com/books?id=9NGWq3K1RwUC&pg=PA18.

3. Gamma et al. 1995, p. 17.
4. Gamma et al. 1995, p. 14.
5. "3. Data model" (https://docs.python.org/reference/datamodel.html). *The Python Language Reference*. Python Software Foundation. Retrieved 2012-04-26.
6. Booch 1994, p. 86-88.
7. "Classes (I)" (http://www.cplusplus.com/doc/tutorial/classes/). *C++ Language Tutorial*. cplusplus.com. Retrieved 2012-04-29.
8. "Classes (II)" (http://www.cplusplus.com/doc/tutorial/classes2/). *C++ Language Tutorial*. cplusplus.com. Retrieved 2012-04-29.
9. Booch 1994, p. 105.
10. "Controlling Access to Members of a Class" (http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html). *The Java Tutorials*. Oracle. Retrieved 2012-04-19.
11. "OOP08-CPP. Do not return references to private data" (https://www.securecoding.cert.org/confluence/display/cplusplus/OOP08-CPP.+Do+not+return+references+to+private+data). *CERT C++ Secure Coding Standard*. Carnegie Mellon University. 2010-05-10. Retrieved 2012-05-07.
12. Ben-Ari, Mordechai (2007-01-24). "2.2 Identifiers" (http://introcs.cs.princeton.edu/java/11cheatsheet/errors.pdf) (PDF). *Compile and Runtime Errors in Java*. Retrieved 2012-05-07.
13. Wild, Fred. "C++ Interfaces" (http://www.drdobbs.com/cpp/184410630). *Dr. Dobb's*. UBM Techweb. Retrieved 2012-05-02.
14. Thomas; Hunt. "Classes, Objects, and Variables" (http://ruby-doc.org/docs/ProgrammingRuby/html/tut_classes.html). *Programming Ruby: The Pragmatic Programmer's Guide*. Ruby-Doc.org. Retrieved 2012-04-26.
15. "Friendship and inheritance" (http://www.cplusplus.com/doc/tutorial/inheritance/). *C++ Language Tutorial*. cplusplus.com. Retrieved 2012-04-26.
16. Booch 1994, p. 180.
17. Booch 1994, p. 128-129.
18. Booch 1994, p. 112.
19. "Interfaces" (http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html). *The Java Tutorials*. Oracle. Retrieved 2012-05-01.
20. "UML-to-Java transformation in IBM Rational Software Architect editions and related software" (http://www.ibm.com/developerworks/rational/library/08/1202_berfeld/). *ibm.com*. Retrieved 20 December 2013.
21. Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). *Object Oriented Software Engineering*. Addison-Wesley ACM Press. pp. 43–69. ISBN 0-201-54435-0.
22. Knublauch, Holger; Oberle, Daniel; Tetlow, Phil; Wallace, Evan (2006-03-09). "A Semantic Web Primer for Object-Oriented Software Developers" (http://www.w3.org/2001/sw/BestPractices/SE/ODSD/). W3C. Retrieved 2008-07-30.
23. Bell, Donald. "UML Basics: The class diagram" (http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/). *developer Works*. IBM. Retrieved 2012-05-02.
24. Booch 1994, p. 179.
25. "Polymorphism" (http://www.cplusplus.com/doc/tutorial/polymorphism/). *C++ Language Tutorial*. cplusplus.com. Retrieved 2012-05-02.
26. "Abstract Methods and Classes" (http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html). *The Java Tutorials*. Oracle. Retrieved 2012-05-02.
27. "Class Abstraction" (http://php.net/manual/en/language.oop5.abstract.php). *PHP Manual*. The PHP Group. Retrieved 2012-05-02.
28. "Interfaces (C# Programming Guide)" (http://msdn.microsoft.com/en-us/library/ms173156.aspx). *C# Programming Guide*. Microsoft. Retrieved 2013-08-15.
29. "Inheritance (C# Programming Guide)" (http://msdn.microsoft.com/en-us/library/ms173149.aspx). *C# Programming Guide*. Microsoft. Retrieved 2012-05-02.
30. "Nested classes (C++ only)" (http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr061.htm). *XL C/C++ V8.0 for AIX*. IBM. Retrieved 2012-05-07.
31. "Local type names (C++ only)" (http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr063.htm). *XL C/C++ V8.0 for AIX*. IBM. Retrieved 2012-05-07.
32. "Local classes (C++ only)" (http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8a.doc%2Flanguage%2Fref%2Fcplr062.htm). *XL C/C++ V8.0 for AIX*. IBM. Retrieved 2012-05-07.
33. Booch 1994, p. 133-134.
34. "13 Classes and metaclasses" (http://pharo.gforge.inria.fr/PBE1/PBE1ch14.html). *pharo.gforge.inria.fr*. Retrieved 2016-10-31.
35. Thomas; Hunt. "Classes and Objects" (http://www.ruby-doc.org/docs/ProgrammingRuby/html/classes.html). *Programming Ruby: The Pragmatic Programmer's Guide*. Ruby-Doc.org. Retrieved 2012-05-08.

36. Booch 1994, p. 134.
37. "MOP: Concepts" (http://www.alu.org/mop/concepts.html#introduction). *The Common Lisp Object System MetaObject Protocol*. Association of Lisp Users. Retrieved 2012-05-08.
38. "sealed (C# Reference)" (http://msdn.microsoft.com/en-us/library/ms173149.aspx). *C# Reference*. Microsoft. Retrieved 2012-05-08.
39. "Writing Final Classes and Methods" (http://docs.oracle.com/javase/tutorial/java/IandI/final.html). *The Java Tutorials*. Oracle. Retrieved 2012-05-08.
40. "PHP: Final Keyword" (http://php.net/manual/en/language.oop5.final.php). *PHP Manual*. The PHP Group. Retrieved 2014-08-21.
41. "String (Java Platform SE 7)" (http://docs.oracle.com/javase/7/docs/api/java/lang/String.html). *Java Platform, Standard Edition 7: API Specification*. Oracle. Retrieved 2012-05-08.
42. "9. Classes" (https://docs.python.org/3.3/tutorial/classes.html). *The Python Tutorial*. Python.org. Retrieved 3 March 2018. "As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation."
43. mairaw; BillWagner; tompratt-AQ (2015-09-19), "Partial Classes and Methods" (https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods), *C# Programming Guide*, Microsoft, retrieved 2018-08-08
44. Apple (2014-09-17), "Customizing Existing Classes" (https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/CustomizingExistingClasses/CustomizingExistingClasses.html#//apple_ref/doc/uid/TP40011210-CH6-SW1), *Programming with Objective-C*, Apple, retrieved 2018-08-08
45. "Static Classes and Static Class Members (C# Programming Guide)" (http://msdn.microsoft.com/en-us/library/79b3xss3(v=vs.100).aspx). *C# Programming Guide*. Microsoft. Retrieved 2012-05-08.
46. "What is an Object?" (http://docs.oracle.com/javase/tutorial/java/concepts/object.html). *oracle.com*. Oracle Corporation. Retrieved 13 December 2013.
47. Booch, Grady; Robert A. Maksimchuk; Michael W. Engle; Bobbi J. Young Ph.D.; Jim Conallen; Kelli A. Houston (April 30, 2007). *Object-Oriented Analysis and Design with Applications* (http://my.safaribooksonline.com/book/software-engineering-and-development/object/9780201895513). Addison-Wesley Professional. pp. 1–28. ISBN 978-0-201-89551-3. Retrieved 20 December 2013. "There are fundamental limiting factors of human cognition; we can address these constraints through the use of decomposition, abstraction, and hierarchy."
48. Jacobsen, Ivar; Magnus Christerson; Patrik Jonsson; Gunnar Overgaard (1992). *Object Oriented Software Engineering*. Addison-Wesley ACM Press. ISBN 0-201-54435-0.

References

- Booch, Grady (1994). *Objects and Design with Applications, Second Edition*. Benjamin/Cummings.
- Gamma; Helm; Johnson; Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.
- Bruce, Kim B. (2002). *Foundations of Object-Oriented Languages: Types and Semantics* (http://mitpress.mit.edu/books/foundations-object-oriented-languages). Cambridge, MA: MIT Press. ISBN 978-0-262-02523-2.

Further reading

- Abadi; Cardelli: A Theory of Objects (http://lucacardelli.name/TheoryOfObjects.html)
- ISO/IEC 14882:2003 Programming Language C++, International standard (http://www.open-std.org/jtc1/sc22/wg21/)
- Class Warfare: Classes vs. Prototypes (http://www.laputan.org/reflection/warfare.html), by Brian Foote
- Meyer, B.: "Object-oriented software construction", 2nd edition, Prentice Hall, 1997, ISBN 0-13-629155-4
- Rumbaugh et al.: "Object-oriented modeling and design", Prentice Hall, 1991, ISBN 0-13-630054-5

External links

- Dias, Tiago (October 2006). "Programming demo - .NET using Partial Types for better code" (https://www.youtube.com/watch?v=oaw8K8GNhAI). *Hyper/Net*. Youtube.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Class_(computer_programming)&oldid=860065842"

This page was last edited on 18 September 2018, at 03:03 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.