WIKIPEDIA

Inheritance (object-oriented programming)

In object-oriented programming, **inheritance** is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones (super class or base class) and forming them into a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all the properties and behaviors of the parent object (except: constructors, destructor, overloaded operators and friend functions of the base class). Inheritance allows programmers to create classes that are built upon existing classes,[1] to specify a new implementation while maintaining the same behaviors (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a directed graph. Inheritance was invented in 1969 for Simula.[2]

An inherited class is called a **subclass** of its parent class or super class. The term "inheritance" is loosely used for both class-based and prototype-based programming, but in narrow use the term is reserved for class-based programming (one class *inherits from* another), with the corresponding technique in prototype-based programming being instead called *delegation* (one object *delegates to* another).

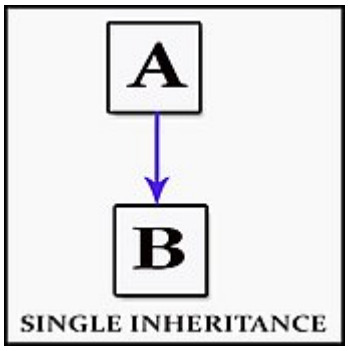
Inheritance should not be confused with subtyping.[3][4] In some languages inheritance and subtyping agree,[a] whereas in others they differ; in general, subtyping establishes an is-a relationship, whereas inheritance only reuses implementation and establishes a syntactic relationship, not necessarily a semantic relationship (inheritance does not ensure behavioral subtyping). To distinguish these concepts, subtyping is also known as *interface inheritance*, whereas inheritance as defined here is known as *implementation inheritance* or *code inheritance*.[5] Still, inheritance is a commonly used mechanism for establishing subtype relationships.[6]

Inheritance is contrasted with object composition, where one object *contains* another object (or objects of one class contain objects of another class); see composition over inheritance. Composition implements a has-a relationship, in contrast to the is-a relationship of subtyping.

Contents

Types

There are various types of inheritance, based on paradigm and specific language.[7]



Single inheritance

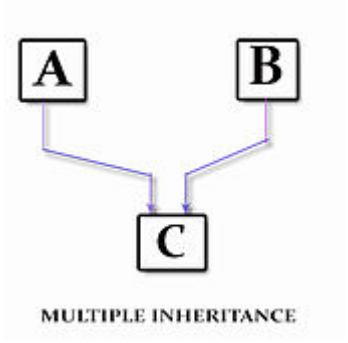
Single inheritance

where subclasses inherit the features of one superclass. A class acquires the properties of another class.

Multiple inheritance

where one class can have more than one superclass and inherit features from all parent classes.

> "Multiple Inheritance (object-oriented programming) was widely supposed to be very difficult to implement efficiently. For example, in a summary of C++ in his book on objective C Brd.Cox actually claimed that adding Multiple inheritance to C++ was impossible. Thus, multiple inheritance seemed more of a challenge. Since I had considered multiple inheritance as early as 1982 and found a simple and efficient implementation technique in 1984. I couldn't resist the challenge. I suspect this to be the only case in which fashion affected the sequence of events."[8]
>
> — Bjarne Stroustrup



Multiple inheritance

In JDK 1.8, Java now has support for multiple inheritance[9].

Multilevel inheritance

where a subclass is inherited from another subclass. It is not uncommon that a class is derived from another derived class as shown in the figure "Multilevel inheritance".

The class *A* serves as a *base class* for the *derived class B*, which in turn serves as a *base class* for the *derived class C*. The class *B* is known as *intermediate* base class because it provides a link for the inheritance between *A* and *C*. The chain *ABC* is known as *inheritance path*.

A derived class with multilevel inheritance is declared as follows:

```
Class A(...);      //Base class
Class B : public A(...);   //B derived from A
Class C : public B(...);   //C derived from B
```

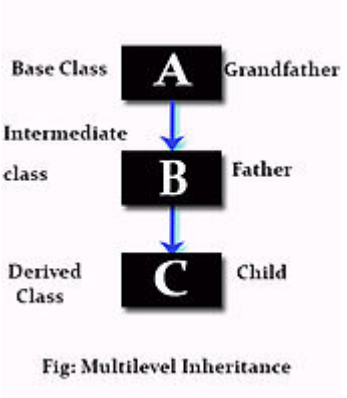
This process can be extended to any number of levels.

Hierarchical inheritance

where one class serves as a superclass (base class) for more than one sub class.

Hybrid inheritance

a mix of two or more of the above types of inheritance.



Multilevel inheritance

Subclasses and superclasses

Subclasses, *derived classes*, *heir classes*, or *child classes* are modular derivative classes that inherits one or more language entities from one or more other classes (called *superclass*, *base classes*, or *parent classes*). The semantics of class inheritance vary from language to language, but commonly the subclass automatically inherits the instance variables and member functions of its superclasses.

The general form of defining a derived class is:[10]

```
class SubClass: visibility SuperClass
{
    // subclass members
};
```

- The colon indicates that the subclass inherits from the superclass. The visibility is optional and, if present, may be either *private* or *public*. The default visibility is *private*. Visibility specifies whether the features of the base class are *privately derived* or *publicly derived*.

Some languages support also the inheritance of other constructs. For example, in Eiffel, contracts that define the specification of a class are also inherited by heirs. The superclass establishes a common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement. The software inherited by a subclass is considered reused in the subclass. A reference to an instance of a class may actually be referring to one of its subclasses. The actual class of the object being referenced is impossible to predict at compile-time. A uniform interface is used to invoke the member functions of objects of a number of different classes. Subclasses may replace superclass functions with entirely new functions that must share the same method signature.

Non-subclassable classes

In some languages a class may be declared as non-subclassable by adding certain class modifiers to the class declaration. Examples include the `final` keyword in Java and C++11 onwards or the `sealed` keyword in C#. Such modifiers are added to the class declaration before the `class` keyword and the class identifier declaration. Such non-subclassable classes restrict reusability, particularly when developers only have access to precompiled binaries and not source code.

A non-subclassable class has no subclasses, so it can be easily deduced at compile time that references or pointers to objects of that class are actually referencing instances of that class and not instances of subclasses (they don't exist) or instances of superclasses (upcasting a reference type violates the type system). Because the exact type of the object being referenced is known before execution, early binding (also called static dispatch) can be used instead of late binding (also called dynamic dispatch), which requires one or more virtual method table lookups depending on whether multiple inheritance or only single inheritance are supported in the programming language that is being used.

Non-overridable methods

Just as classes may be non-subclassable, method declarations may contain method modifiers that prevent the method from being overridden (i.e. replaced with a new function with the same name and type signature in a subclass). A private method is unoverridable simply because it is not accessible by classes other than the class it is a member function of (this is not true for C++, though). A `final` method in Java, a `sealed` method in C# or a `frozen` feature in Eiffel cannot be overridden.

Virtual methods

If the superclass method is a virtual method, then invocations of the superclass method will be dynamically dispatched. Some languages require that methods be specifically declared as virtual (e.g. C++), and in others, all methods are virtual (e.g. Java). An invocation of a non-virtual method will always be statically dispatched (i.e. the address of the function call is determined at compile-time). Static dispatch is faster than dynamic dispatch and allows optimisations such as inline expansion.

Visibility of inherited members

Source:[11]

Base class visibility	Derived class visibility		
	Public derivation	**Private derivation**	**Protected derivation**
▪ Private → ▪ Protected → ▪ Public →	▪ Not inherited ▪ Protected ▪ Public	▪ Not inherited ▪ Private ▪ Private	▪ Not inherited ▪ Protected ▪ Protected

Applications

Inheritance is used to co-relate two or more classes to each other.

Overriding

Many object-oriented programming languages permit a class or object to replace the implementation of an aspect—typically a behavior—that it has inherited. This process is usually called *overriding*. Overriding introduces a complication: which version of the behavior does an instance of the inherited class use—the one that is part of its own class, or the one from the parent (base) class? The answer varies between programming languages, and some languages provide the ability to indicate that a particular behavior is not to be overridden and should behave as defined by the base class. For instance, in C#, the base method or property can only be overridden in a subclass if it is marked with the virtual, abstract, or override modifier.[12] An alternative to overriding is hiding the inherited code.

Code reuse

Implementation inheritance is the mechanism whereby a subclass re-uses code in a base class. By default the subclass retains all of the operations of the base class, but the subclass may override some or all operations, replacing the base-class implementation with its own.

In the following Python example, subclasses `SquareSumComputer` and `CubeSumComputer` override the `transform()` method of the base class `SumComputer`. The base class comprises operations to compute the sum of the squares between two integers. The subclass re-uses all of the functionality of the base class with the exception of the operation that transforms a number into its square, replacing it with an operation that transforms a number into its square and cube respectively. The subclasses therefore compute the sum of the squares/cubes between two integers.

Below is an example of Python

```
class SumComputer(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def transform(self, x):
        raise NotImplementedError

    def inputs(self):
        return range(self.a, self.b)

    def compute(self):
        return sum(self.transform(value) for value in self.inputs())

class SquareSumComputer(SumComputer):
    def transform(self, x):
        return x * x

class CubeSumComputer(SumComputer):
    def transform(self, x):
        return x * x * x
```

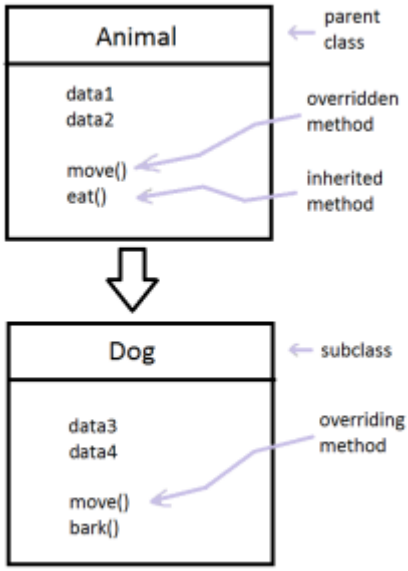


Illustration of method overriding

In most quarters, class inheritance for the sole purpose of code reuse has fallen out of favor. The primary concern is that implementation inheritance does not provide any assurance of polymorphic substitutability—an instance of the reusing class cannot necessarily be substituted for an instance of the inherited class. An alternative technique, explicit delegation, requires more programming effort, but avoids the substitutability issue. In C++ private inheritance can be used as a form of *implementation inheritance* without substitutability. Whereas public inheritance represents an "is-a" relationship and delegation represents a "has-a" relationship, private (and protected) inheritance can be thought of as an "is implemented in terms of" relationship.[13]

Another frequent use of inheritance is to guarantee that classes maintain a certain common interface; that is, they implement the same methods. The parent class can be a combination of implemented operations and operations that are to be implemented in the child classes. Often, there is no interface change between the supertype and subtype- the child implements the behavior described instead of its parent class.[14]

Inheritance vs subtyping

Inheritance is similar to but distinct from subtyping.[15] Subtyping enables a given type to be substituted for another type or abstraction, and is said to establish an *is-a* relationship between the subtype and some existing abstraction, either implicitly or explicitly, depending on language support. The relationship can be expressed explicitly via inheritance in languages that support inheritance as a subtyping mechanism. For example, the following C++ code establishes an explicit inheritance relationship between classes *B* and *A*, where *B* is both a subclass and a subtype of *A*, and can be used as an *A* wherever a *B* is specified (via a reference, a pointer or the object itself).

```
class A {
 public:
  void DoSomethingALike() const {}
};

class B : public A {
 public:
  void DoSomethingBLike() const {}
};

void UseAnA(const A& a) {
  a.DoSomethingALike();
}

void SomeFunc() {
  B b;
  UseAnA(b);  // b can be substituted for an A.
}
```

In programming languages that do not support inheritance as a subtyping mechanism, the relationship between a base class and a derived class is only a relationship between implementations (a mechanism for code reuse), as compared to a relationship between types. Inheritance, even in programming languages that support inheritance as a subtyping mechanism, does not necessarily entail behavioral subtyping. It is entirely possible to derive a class whose object will behave incorrectly when used in a context where the parent class is expected; see the Liskov substitution principle. [16] (Compare connotation/denotation.) In some OOP languages, the notions of code reuse and subtyping coincide because the only way to declare a subtype is to define a new class that inherits the implementation of another.

Design constraints

Using inheritance extensively in designing a program imposes certain constraints.

For example, consider a class `Person` that contains a person's name, date of birth, address and phone number. We can define a subclass of `Person` called `Student` that contains the person's grade point average and classes taken, and another subclass of `Person` called `Employee` that contains the person's job-title, employer, and salary.

In defining this inheritance hierarchy we have already defined certain restrictions, not all of which are desirable:

Singleness

Using single inheritance, a subclass can inherit from only one superclass. Continuing the example given above, `Person` can be either a `Student` or an `Employee`, but not both. Using multiple inheritance partially solves this problem, as one can then define a `StudentEmployee` class that inherits from both `Student` and `Employee`. However, in most implementations, it can still inherit from each superclass only once, and thus, does not support cases in which a student has two jobs or attends two institutions. The inheritance model available in Eiffel makes this possible through support for repeated inheritance.

Static

The inheritance hierarchy of an object is fixed at instantiation when the object's type is selected and does not change with time. For example, the inheritance graph does not allow a `Student` object to become a `Employee` object while retaining the state of its `Person` superclass. (This kind of behavior, however, can be achieved with the decorator pattern.) Some have criticized inheritance, contending that it locks developers into their original design standards.[17]

Visibility

Whenever client code has access to an object, it generally has access to all the object's superclass data. Even if the superclass has not been declared public, the client can still cast the

object to its superclass type. For example, there is no way to give a function a pointer to a `Student`'s grade point average and transcript without also giving that function access to all of the personal data stored in the student's `Person` superclass. Many modern languages, including C++ and Java, provide a "protected" access modifier that allows subclasses to access the data, without allowing any code outside the chain of inheritance to access it.

The composite reuse principle is an alternative to inheritance. This technique supports polymorphism and code reuse by separating behaviors from the primary class hierarchy and including specific behavior classes as required in any business domain class. This approach avoids the static nature of a class hierarchy by allowing behavior modifications at run time and allows one class to implement behaviors buffet-style, instead of being restricted to the behaviors of its ancestor classes.

Issues and alternatives

Implementation inheritance is controversial among programmers and theoreticians of object-oriented programming since at least the 1990s. Among them are the authors of *Design Patterns*, who advocate interface inheritance instead, and favor composition over inheritance. For example, the decorator pattern (as mentioned above) has been proposed to overcome the static nature of inheritance between classes. As a more fundamental solution to the same problem, role-oriented programming introduces a distinct relationship, *played-by*, combining properties of inheritance and composition into a new concept.

According to Allen Holub, the main problem with implementation inheritance is that it introduces unnecessary coupling in the form of the "fragile base class problem":[5] modifications to the base class implementation can cause inadvertent behavioral changes in subclasses. Using interfaces avoids this problem because no implementation is shared, only the API.[17] Another way of stating this is that "inheritance breaks encapsulation".[18] The problem surfaces clearly in open object-oriented systems such as frameworks, where client code is expected to inherit from system-supplied classes and then substituted for the system's classes in its algorithms.[5]

Reportedly, Java inventor James Gosling has spoken against implementation inheritance, stating that he would not include it if he were to redesign Java.[17] Language designs that decouple inheritance from subtyping (interface inheritance) appeared as early as 1990;[19] a modern example of this is the Go programming language.

Complex inheritance, or inheritance used within an insufficiently mature design, may lead to the yo-yo problem. When inheritance was used as a primary approach to structure code in a system in the late 90's, developers naturally started to break code into multiple layers of inheritance as the system functionality grew. If a development team combined multiple layers of inheritance with the single responsibility principle it created many super thin layers of code, many which would only have 1 or 2 lines of code in each layer. Before teams learned the hard way that 2 or 3 layers was optimum number of layers balancing the benefit of code reuse against the complexity increase with each layer, it was not uncommon to work on inheritance frameworks with 10 and up to 30 layers. For instance 30 layers made debugging a significant challenge simply to know which layer needed debugged. PowerBuilder built one of the best code library that primarily used inheritance, it was built with 3 to 4 layers. The number of layers in an inheritance library is critical and must be kept at or below 4 layers otherwise the library becomes too complex and time consuming to use.

Another issue with inheritance is that subclasses must be defined in code, which means that program users cannot add new subclasses. Other design patterns (such as Entity–component–system) allow program users to define variations of an entity at runtime.

See also

- Archetype pattern
- Circle-ellipse problem
- Defeasible reasoning
- Interface (computing)
- Method overriding
- Mixin
- Polymorphism (computer science)
- Protocol
- Role-oriented programming
- The Third Manifesto
- Trait (computer programming)
- Virtual inheritance

Notes

a. This is generally true only in statically-typed class-based OO languages, such as C++, C#, Java, and Scala.

References

1. Johnson, Ralph (August 26, 1991). "Designing Reusable Classes" (https://www.cse.msu.edu/~cse870/Input/SS2002/MiniProject/Sources/DRC.pdf) (PDF). *www.cse.msu.edu*.
2. Mike Mintz, Robert Ekendahl (2006). *Hardware Verification with C++: A Practitioner's Handbook*. United States of America: Springer. p. 22. ISBN 978-0-387-25543-9.
3. Cook, William R.; Hill, Walter; Canning, Peter S. (1990). *Inheritance is not subtyping*. Proc. 17th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL). pp. 125–135. CiteSeerX 10.1.1.102.8635 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.8635). doi:10.1145/96709.96721 (https://doi.org/10.1145%2F96709.96721). ISBN 0-89791-343-4.
4. Cardelli, Luca (1993). *Typeful Programming* (Technical report). Digital Equipment Corporation. p. 32–33. SRC Research Report 45.
5. Mikhajlov, Leonid; Sekerinski, Emil (1998). *A study of the fragile base class problem* (http://extras.springer.com/2000/978-3-540-67660-7/papers/1445/14450355.pdf) (PDF). Proc. 12th European Conf. on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science. **1445**. pp. 355–382. doi:10.1007/BFb0054099 (https://doi.org/10.1007%2FBFb0054099). ISBN 978-3-540-64737-9.
6. Tempero, Ewan; Yang, Hong Yul; Noble, James (2013). *What programmers do with inheritance in Java* (https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/TemperoYangNobleECOOP2013-pre.pdf) (PDF). ECOOP 2013–Object-Oriented Programming. pp. 577–601.
7. "C++ Inheritance" (https://www.cs.nmsu.edu/~rth/cs/cs177/map/inheritd.html). *www.cs.nmsu.edu*.
8. Bjarne Stroustrup. *The Design and Evolution of C++*. p. 417.
9. "Java Multiple Inheritance" (http://howtodoinjava.com/object-oriented/multiple-inheritance-in-java/). *HowToDoInJava*. Retrieved 28 June 2017.
10. Herbert Schildt (2003). *The complete reference C++*. Tata McGrawhill Education Private Limited. p. 417. ISBN 978-0-07-053246-5.
11. E Balagurusamy (2010). *Object Orientedprogramming With C++*. Tata McGrawhill Education Pvt. Ltd. p. 213. ISBN 978-0-07-066907-9.
12. override(C# Reference) (http://msdn.microsoft.com/en-us/library/ebca9ah3.aspx)
13. "GotW #60: Exception-Safe Class Design, Part 2: Inheritance" (http://www.gotw.ca/gotw/060.htm). Gotw.ca. Retrieved 2012-08-15.
14. Dr. K. R. Venugopal, Rajkumar Buyya (2013). *Mastering C++*. Tata McGrawhill Education Private Limited. p. 609. ISBN 9781259029943.

15. Cook, Hill & Canning 1990.
16. Mitchell, John (2002). "10 "Concepts in object-oriented languages" ". *Concepts in programming language*. Cambridge, UK: Cambridge University Press. p. 287. ISBN 978-0-521-78098-8.
17. Holub, Allen (1 August 2003). "Why extends is evil" (http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html). Retrieved 10 March 2015.
18. Seiter, Linda M.; Palsberg, Jens; Lieberherr, Karl J. (1996). "Evolution of object behavior using context relations" (http://www.ccs.neu.edu/research/demeter/papers/context-journal/_context.html). *ACM SIGSOFT Software Engineering Notes*. **21** (6): 46. CiteSeerX 10.1.1.36.5053 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.5053). doi:10.1145/250707.239108 (https://doi.org/10.1145%2F250707.239108).
19. America, Pierre (1991). *Designing an object-oriented programming language with behavioural subtyping* (https://www.researchgate.net/profile/Pierre_America/publication/221501695_Designing_an_Object-Oriented_Programming_Language_with_Behavioural_Subtyping/links/00b7d52e0f188abba4000000.pdf) (PDF). REX School/Workshop on the Foundations of Object-Oriented Languages. Lecture Notes in Computer Science. **489**. pp. 60–90. doi:10.1007/BFb0019440 (https://doi.org/10.1007%2FBFb0019440). ISBN 978-3-540-53931-5.

Further reading

- Object-Oriented Software Construction*, Second Edition, by Bertrand Meyer, Prentice Hall, 1997, ISBN 0-13-629155-4, Chapter 24: Using Inheritance Well.*
- Implementation Inheritance Is Evil (https://medium.com/@wrong.about/inheritance-based-on-internal-structure-is-evil-7474cc8e64dc)