WIKIPEDIA
The Free Encyclopedia

Multiple inheritance

Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit features from more than one parent object or parent class. It is distinct from single inheritance, where an object or class may only inherit from one particular object or class.

Multiple inheritance has been a controversial issue for many years,[1][2] with opponents pointing to its increased complexity and ambiguity in situations such as the "diamond problem", where it may be ambiguous as to which parent class a particular feature is inherited from if more than one parent class implements said feature. This can be addressed in various ways, including using virtual inheritance.[3] Alternate methods of object composition not based on inheritance such as mixins and traits have also been proposed to address the ambiguity.

Details

In object-oriented programming (OOP), *inheritance* describes a relationship between two classes in which one class (the *child* class) *subclasses* the *parent* class. The child inherits methods and attributes of the parent, allowing for shared functionality. For example, one might create a variable class *Mammal* with features such as eating, reproducing, etc.; then define a child class *Cat* that inherits those features without having to explicitly program them, while adding new features like *chasing mice*.

Multiple inheritance allows programmers to use more than one totally orthogonal hierarchy simultaneously, such as allowing *Cat* to inherit from *Cartoon character* and *Pet* and *Mammal* and access features from within all of those classes.

Implementations

Languages that support multiple inheritance include: C++, Common Lisp (via Common Lisp Object System (CLOS)), EuLisp (via The EuLisp Object System TELOS), Curl, Dylan, Eiffel, Logtalk, Object REXX, Scala (via use of mixin classes), OCaml, Perl, POP-11, Python, R, Raku, and Tcl (built-in from 8.6 or via Incremental Tcl (Incr Tcl) in earlier versions[4][5]).

IBM System Object Model (SOM) runtime supports multiple inheritance, and any programming language targeting SOM can implement new SOM classes inherited from multiple bases.

Some object-oriented languages, such as Swift, Java, Fortran since its 2003 revision, C#, and Ruby implement *single inheritance*, although protocols, or *interfaces,* provide some of the functionality of true multiple inheritance.

PHP uses traits classes to inherit specific method implementations. Ruby uses modules to inherit multiple methods.

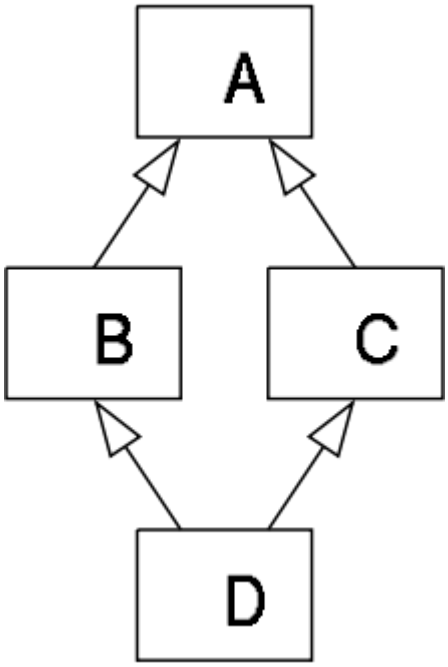
The diamond problem

The "**diamond problem**" (sometimes referred to as the "Deadly Diamond of Death"[6]) is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then

which version of the method does D inherit: that of B, or that of C?

For example, in the context of GUI software development, a class `Button` may inherit from both classes `Rectangle` (for appearance) and `Clickable` (for functionality/input handling), and classes `Rectangle` and `Clickable` both inherit from the `Object` class. Now if the `equals` method is called for a `Button` object and there is no such method in the `Button` class but there is an overridden `equals` method in `Rectangle` or `Clickable` (or both), which method should be eventually called?

It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation. In this case, class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape.



A diamond class inheritance diagram.

Mitigation

Languages have different ways of dealing with these problems of repeated inheritance.

- C# (since C# 8.0) allows default interface method implementation, causing a class `A`, implementing interfaces `Ia` and `Ib` with similar methods having default implementations, to have two "inherited" methods with the same signature, causing the diamond problem. It is mitigated either by having `A` to implement the method itself, hence removing ambiguity, or forcing the caller to first cast the `A` object to the appropriate interface to use its default implementation of that method (e.g. `((Ia) aInstance).Method();`).
- C++ by default follows each inheritance path separately, so a `D` object would actually contain two separate `A` objects, and uses of `A`'s members have to be properly qualified. If the inheritance from `A` to `B` and the inheritance from `A` to `C` are both marked "`virtual`" (for example, "`class B : virtual public A`"), C++ takes special care to only create one `A` object, and uses of `A`'s members work correctly. If virtual inheritance and nonvirtual inheritance are mixed, there is a single virtual `A`, and a nonvirtual `A` for each nonvirtual inheritance path to `A`. C++ requires stating explicitly which parent class the feature to be used is invoked from i.e. `Worker::Human.Age`. C++ does not support explicit repeated inheritance since there would be no way to qualify which superclass to use (i.e. having a class appear more than once in a single derivation list [class Dog : public Animal, Animal]). C++ also allows a single instance of the multiple class to be created via the virtual inheritance mechanism (i.e. `Worker::Human` and `Musician::Human` will reference the same object).
- Common Lisp CLOS attempts to provide both reasonable default behavior and the ability to override it. By default, to put it simply, the methods are sorted in `D`, `B`, `C`, `A`, when B is written before C in the class definition. The method with the most specific argument classes is chosen (D>(B,C)>A) ; then in the order in which parent classes are named in the subclass definition (B>C). However, the programmer can override this, by giving a specific method resolution order or stating a rule for combining methods. This is called method combination, which may be fully controlled. The MOP (metaobject protocol) also provides means to modify the inheritance, dynamic dispatch, class

instantiation, and other internal mechanisms without affecting the stability of the system.

- Curl allows only classes that are explicitly marked as *shared* to be inherited repeatedly. Shared classes must define a *secondary constructor* for each regular constructor in the class. The regular constructor is called the first time the state for the shared class is initialized through a subclass constructor, and the secondary constructor will be invoked for all other subclasses.
- In Eiffel, the ancestors' features are chosen explicitly with select and rename directives. This allows the features of the base class to be shared between its descendants or to give each of them a separate copy of the base class. Eiffel allows explicit joining or separation of features inherited from ancestor classes. Eiffel will automatically join features together, if they have the same name and implementation. The class writer has the option to rename the inherited features to separate them. Multiple inheritance is a frequent occurrence in Eiffel development; most of the effective classes in the widely used EiffelBase library of data structures and algorithms, for example, have two or more parents.[7]
- Go prevents the diamond problem at compile time. If a structure `D` embeds two structures `B` and `C` which both have a method `F()`, thus satisfying an interface `A`, the compiler will complain about an "ambiguous selector" if `D.F()` is called, or if an instance of `D` is assigned to a variable of type `A`. `B` and `C`'s methods can be called explicitly with `D.B.F()` or `D.C.F()`.
- Java 8 introduces default methods on interfaces. If `A`, `B`, `C` are interfaces, `B`, `C` can each provide a different implementation to an abstract method of `A`, causing the diamond problem. Either class `D` must reimplement the method (the body of which can simply forward the call to one of the super implementations), or the ambiguity will be rejected as a compile error.[8] Prior to Java 8, Java was not subject to the Diamond problem risk, because it did not support multiple inheritance and interface default methods were not available.
- JavaFX Script in version 1.2 allows multiple inheritance through the use of mixins. In case of conflict, the compiler prohibits the direct usage of the ambiguous variable or function. Each inherited member can still be accessed by casting the object to the mixin of interest, e.g. `(individual as Person).printInfo();`.
- Kotlin allows multiple inheritance of Interfaces, however, in a Diamond problem scenario, the child class must override the method that causes the inheritance conflict and specify which parent class implementation should be used. eg `super<ChosenParentInterface>.someMethod()`
- Logtalk supports both interface and implementation multi-inheritance, allowing the declaration of method *aliases* that provide both renaming and access to methods that would be masked out by the default conflict resolution mechanism.
- In OCaml, parent classes are specified individually in the body of the class definition. Methods (and attributes) are inherited in the same order, with each newly inherited method overriding any existing methods. OCaml chooses the last matching definition of a class inheritance list to resolve which method implementation to use under ambiguities. To override the default behavior, one simply qualifies a method call with the desired class definition.
- Perl uses the list of classes to inherit from as an ordered list. The compiler uses the first method it finds by depth-first searching of the superclass list or using the C3 linearization of the class hierarchy. Various extensions provide alternative class

composition schemes. The order of inheritance affects the class semantics. In the above ambiguity, class `B` and its ancestors would be checked before class `C` and its ancestors, so the method in `A` would be inherited through `B`. This is shared with Io and Picolisp. In Perl, this behavior can be overridden using the `mro` or other modules to use C3 linearization or other algorithms.[9]

- Python has the same structure as Perl, but, unlike Perl, includes it in the syntax of the language. The order of inheritance affects the class semantics. Python had to deal with this upon the introduction of new-style classes, all of which have a common ancestor, `object`. Python creates a list of classes using the C3 linearization (or Method Resolution Order (MRO)) algorithm. That algorithm enforces two constraints: children precede their parents and if a class inherits from multiple classes, they are kept in the order specified in the tuple of base classes (however in this case, some classes high in the inheritance graph may precede classes lower in the graph[10]). Thus, the method resolution order is: `D`, `B`, `C`, `A`.[11]
- Ruby classes have exactly one parent but may also inherit from multiple *modules;* ruby class definitions are executed, and the (re)definition of a method obscures any previously existing definition at the time of execution. In the absence of runtime metaprogramming this has approximately the same semantics as rightmost depth first resolution.
- Scala allows multiple instantiation of *traits*, which allows for multiple inheritance by adding a distinction between the class hierarchy and the trait hierarchy. A class can only inherit from a single class, but can mix-in as many traits as desired. Scala resolves method names using a right-first depth-first search of extended 'traits', before eliminating all but the last occurrence of each module in the resulting list. So, the resolution order is: [`D`, `C`, `A`, `B`, `A`], which reduces down to [`D`, `C`, `B`, `A`].
- Tcl allows multiple parent classes; the order of specification in the class declaration affects the name resolution for members using the C3 linearization algorithm.[12]

Languages that allow only single inheritance, where a class can only derive from one base class, do not have the diamond problem. The reason for this is that such languages have at most one implementation of any method at any level in the inheritance chain regardless of the repetition or placement of methods. Typically these languages allow classes to implement multiple protocols, called interfaces in Java. These protocols define methods but do not provide concrete implementations. This strategy has been used by ActionScript, C#, D, Java, Nemerle, Object Pascal, Objective-C, Smalltalk, Swift and PHP.[13] All these languages allow classes to implement multiple protocols.

Moreover, Ada, C#, Java, Object Pascal, Objective-C, Swift and PHP allow multiple-inheritance of interfaces (called protocols in Objective-C and Swift). Interfaces are like abstract base classes that specify method signatures without implementing any behaviour. ("Pure" interfaces such as the ones in Java up to version 7 do not permit any implementation or instance data in the interface.) Nevertheless, even when several interfaces declare the same method signature, as soon as that method is implemented (defined) anywhere in the inheritance chain, it overrides any implementation of that method in the chain above it (in its superclasses). Hence, at any given level in the inheritance chain, there can be at most one implementation of any method. Thus, single-inheritance method implementation does not exhibit the Diamond Problem even with multiple-

inheritance of interfaces. With the introduction of default implementation for interfaces in Java 8 and C# 8, it is still possible to generate a Diamond Problem, although this will only appear as a compile-time error.

See also

- Directed graph
- Nixon diamond

References

1. Cargill, T. A. (Winter 1991). "Controversy: The Case Against Multiple Inheritance in C++". *Computing Systems*. **4** (1): 69–82.
2. Waldo, Jim (Spring 1991). "Controversy: The Case For Multiple Inheritance in C++". *Computing Systems*. **4** (2): 157–171.
3. Schärli, Nathanael; Ducasse, Stéphane; Nierstrasz, Oscar; Black, Andrew. "Traits: Composable Units of Behavior" (http://web.cecs.pdx.edu/~black/publications/TR_CSE_02-012.pdf) (PDF). *Web.cecs.pdx.edu*. Retrieved 2016-10-21.
4. "incr Tcl" (http://blog.tcl.tk/62). *blog.tcl.tk*. Retrieved 2020-04-14.
5. "Introduction to the Tcl Programming Language" (https://www2.lib.uchicago.edu/keith/tcl-course/). *www2.lib.uchicago.edu*. Retrieved 2020-04-14.
6. Martin, Robert C. (1997-03-09). "Java and C++: A critical comparison" (https://web.archive.org/web/20051024230813/http://www.objectmentor.com/resources/articles/javacpp.pdf) (PDF). *Objectmentor.com*. Archived from the original (http://objectmentor.com/resources/articles/javacpp.pdf) (PDF) on 2005-10-24. Retrieved 2016-10-21.
7. "Standard ECMA-367" (http://www.ecma-international.org/publications/standards/Ecma-367.htm). *Ecma-international.org*. Retrieved 2016-10-21.
8. "State of the Lambda" (http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html). *Cr.openjdk.java.net*. Retrieved 2016-10-21.
9. "perlobj" (http://perldoc.perl.org/perlobj.html#Method-Resolution-Order). *perldoc.perl.org*. Retrieved 2016-10-21.
10. Abstract. "The Python 2.3 Method Resolution Order" (https://www.python.org/download/releases/2.3/mro/#examples). *Python.org*. Retrieved 2016-10-21.
11. "Unifying types and classes in Python 2.2" (https://www.python.org/download/releases/2.2.3/descrintro/#mro). *Python.org*. Retrieved 2016-10-21.
12. "Manpage of class" (http://www.tcl.tk/man/itcl3.1/class.n.html). *Tcl.tk*. 1999-11-16. Retrieved 2016-10-21.
13. "Object Interfaces - Manual" (http://php.net/manual/en/language.oop5.interfaces.php). *PHP.net*. 2007-07-04. Retrieved 2016-10-21.

Further reading

- Stroustrup, Bjarne (1999). *Multiple Inheritance for C++*. Proceedings of the Spring 1987 European Unix Users Group Conference (http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.23.4735&rep=rep1&type=pdf)
- Object-Oriented Software Construction*, Second Edition, by Bertrand Meyer, Prentice Hall, 1997, ISBN 0-13-629155-4*
- Eddy Truyen; Wouter Joosen; Bo Nørregaard; Pierre Verbaeten (2004). "A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object

Systems" (http://www.cs.kuleuven.ac.be/~eddy/PUBLICATIONS/DAW2004.pdf) (PDF). *Proceedings of the 2004 Dynamic Aspects Workshop* (103–119).

- Ira R. Forman; Scott Danforth (1999). *Putting Metaclasses to Work*. ISBN 0-201-43305-2.

External links

- Tutorial on inheritance usage in Eiffel (http://docs.eiffel.com/book/method/et-inheritance)
- Tutorial on effective use of multiple inheritance in Python (http://rhettinger.wordpress.com/2011/05/26/super-considered-super/)
- An overview of inheritance in Ocaml (https://web.archive.org/web/20091129231735/http://caml.inria.fr/pub/docs/manual-ocaml/manual005.html#toc26)