Object copy

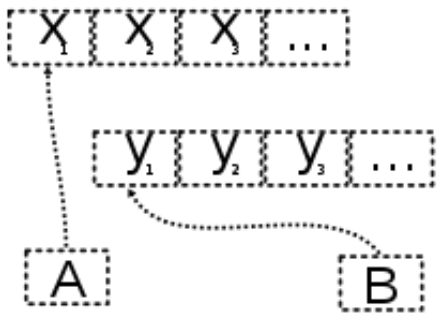
From Wikipedia, the free encyclopedia

An **object copy** is an action in computing where a data object has its attributes copied to another object of the same data type. An object is a composite data type in object-oriented programming languages. The copying of data is one of the most common procedures that occurs in computer programs. An object may be copied to reuse all or part of its data in a new context.

Contents

Methods of copying

The design goal of most objects is to give the resemblance of being made out of one monolithic block even though most are not. As objects are made up of several different parts, copying becomes nontrivial. Several strategies exist to attack this problem.



Example data objects A and B, each consisting of a number of memory blocks.

Consider two objects, A and B, which each refer to two memory blocks x_i and y_i (i = 1, 2,...). Think of A and B as strings and of x_i and y_i (i = 1, 2,...) as the characters they contain. There are different strategies for copying A into B.

Shallow copy

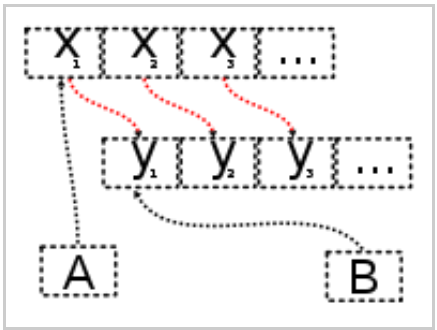
One method of copying an object is the shallow copy. In the process of shallow copying A, B will copy all of A's field values.[1][2][3][4] If the field value is a memory address it copies the memory address, and if the field value is a primitive type it copies the value of the primitive type.

Text

The disadvantage is if you modify the memory address that one of B's fields point to, you are also modifying what A's fields point to.

Deep copy

An alternative is a deep copy. Here the data is actually copied over. The result is different from the result a shallow copy gives. The advantage is that A and B do not depend on each other but at the cost of a slower and more expensive copy.



A deep copy in progress. A deep copy having been completed.

Lazy copy

A lazy copy is a combination of both strategies above. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify an object, it can determine if the data is shared (by examining the counter) and can do a deep copy if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can cause problems.

Lazy copy is related to copy-on-write.

Implementation

Nearly all object-oriented programming languages provide some way to copy objects. As the majority of languages do not provide most objects themselves, the programmer has to define how an object should be copied, just as he or she has to define if two objects are identical or even comparable in the first place. Many languages provide some default behavior.

How copying is solved varies from language to language and what concept of an object it has. The following presents examples for one of the most widely used object-oriented languages, Java, which should cover nearly every way that an object-oriented language can attack this problem.

In Java

Unlike in C++, objects in Java are always accessed indirectly through references. Objects are never created implicitly but instead are always passed or assigned by a reference variable. (Methods in Java are always *pass by value*, however, it is the value of the reference variable that is being passed.)[5] The Java Virtual Machine manages garbage collection so that objects are cleaned up after they are no longer reachable. There is no automatic way to copy any given object in Java.

Copying is usually performed by a clone() method of a class. This method usually, in turn, calls the clone() method of its parent class to obtain a copy, and then does any custom copying procedures. Eventually this gets to the clone() method of `Object` (the uppermost class), which creates a new instance of the same class as the object and copies all the fields to the new instance (a "shallow copy"). If this method is used, the class must implement the `Cloneable` (`http://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html`) marker interface, or

else it will throw a CloneNotSupportedException. After obtaining a copy from the parent class, a class' own clone() method may then provide custom cloning capability, like deep copying (i.e. duplicate some of the structures referred to by the object) or giving the new instance a new unique ID.

The return type of clone() is `Object`, but implementers of a clone method could write the type of the object being cloned instead due to Java's support for covariant return types. One advantage of using clone() is that since it is an overridable method, we can call clone() on any object, and it will use the clone() method of its actual class, without the calling code needing to know what that class is (which would be necessary with a copy constructor).

Another disadvantage is that one often cannot access the clone() method on an abstract type. Most interfaces and abstract classes in Java do not specify a public clone() method. As a result, often the only way to use the clone() method is if the actual class of an object is known, which is contrary to the abstraction principle of using the most generic type possible. For example, if one has a List reference in Java, one cannot invoke clone() on that reference because List specifies no public clone() method. Actual implementations of List like ArrayList and LinkedList all generally have clone() methods themselves, but it is inconvenient and bad abstraction to carry around the actual class type of an object.

Another way to copy objects in Java is to serialize them through the `Serializable (http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)` interface. This is typically used for persistence and wire protocol purposes, but it does create copies of objects and, unlike clone, a deep copy that gracefully handles cycled graphs of objects is readily available with minimal effort from the programmer.

Both of these methods suffer from a notable problem: the constructor is not used for objects copied with clone or serialization. This can lead to bugs with improperly initialized data, prevents the use of `final` member fields, and makes maintenance challenging. Some utilities attempt to overcome these issues by using reflection to deep copy objects, such as the deep-cloning library.[6]

In Eiffel

Runtime objects in Eiffel are accessible either indirectly through references or as *expanded* objects whose fields are embedded within the objects that use them. That is, fields of an object are stored either externally or internally.

The Eiffel class `ANY` contains features for shallow and deep copying and cloning of objects. All Eiffel classes inherit from `ANY`, so these features are available within all classes, and are applicable both to reference and expanded objects.

The `copy` feature effects a shallow, field-by-field copy from one object to another. In this case no new object is created. If `y` were copied to `x`, then the same objects referenced by `y` before the application of `copy`, will also be referenced by `x` after the `copy` feature completes.

To effect the creation of a new object which is a shallow duplicate of `y`, the feature `twin` is used. In this case, a single new object is created with its fields identical to those of the source.

The feature `twin` relies on the feature `copy`, which can be redefined in descendants of `ANY`, if necessary. The result of `twin` is of the anchored type `like Current`.

Deep copying and creating deep twins can be done using the features `deep_copy` and `deep_twin`, again inherited from class `ANY`. These features have the potential to create many new objects, because they duplicate all the objects in an entire object structure. Because new duplicate objects are created instead

of simply copying references to existing objects, deep operations will become a source of performance issues more readily than shallow operations.

In other languages

In C Sharp (C#), rather than using the interface `ICloneable`, a generic extension method can be used to create a deep copy using reflection. This has two advantages: First, it provides the flexibility to copy every object without having to specify each property and variable to be copied manually. Second, because the type is generic, the compiler ensures that the destination object and the source object have the same type.

In Objective-C, the methods `copy` and `mutableCopy` are inherited by all objects and intended for performing copies; the latter is for creating a mutable type of the original object. These methods in turn call the `copyWithZone` and `mutableCopyWithZone` methods, respectively, to perform the copying. An object must implement the corresponding `copyWithZone` method to be copyable.

In OCaml, the library function Oo.copy (http://caml.inria.fr/pub/docs/manual-ocaml/libref/Oo.html#VALcopy) performs shallow copying of an object.

In Python, the library's copy (https://docs.python.org/library/copy.html) module provides shallow copy and deep copy of objects through the `copy()` and `deepcopy()` functions, respectively. Programmers may define special methods `__copy__()` and `__deepcopy__()` in an object to provide custom copying implementation.

In Ruby, all objects inherit two methods for performing shallow copies, clone (http://www.ruby-doc.org/docs/ProgrammingRuby/html/ref_c_object.html#Object.clone) and dup (http://www.ruby-doc.org/docs/ProgrammingRuby/html/ref_c_object.html#Object.dup). The two methods differ in that `clone` copies an object's tainted state, frozen state, and any singleton methods it may have, whereas `dup` copies only its tainted state. Deep copies may be achieved by dumping and loading an object's byte stream or YAML serialization.[1] (http://www.ruby-doc.org/docs/ProgrammingRuby/html/classes.html#S5) Alternatively, you can use the deep_dive gem to do a controlled deep copy of your object graphs. [2] (https://rubygems.org/gems/deep_dive)

In perl, nested structures are stored by the use of references, thus a developer can either loop over the entire structure and re-reference the data or use the `dclone()` function from the module Storable (https://metacpan.org/module/Storable).

In VBA, an assignment of variables of type `Object` is a shallow copy, an assignment for all other types (numeric types, String, user defined types, arrays) is a deep copy. So the keyword `Set` for an assignment signals a shallow copy and the (optional) keyword `Let` signals a deep copy. There is no built-in method for deep copies of Objects in VBA.

See also

- Copy constructor
- Operator overloading
- Reference counting
- Copy-on-write
- Clone (Java method)

References

1. ^ "C++ Shallow vs Deep Copy Explanation" (http://www.fredosaurus.com/notes-cpp/oop-condestructors/shallowdeepcopy.html).
2. ^ ".NET Shallow vs Deep Copy Explanation" (http://www.codeproject.com/Articles/28952/Shallow-Copy-vs-Deep-Copy-in-NET).
3. ^ "Java Shallow vs Deep Copy Explanation" (http://javapapers.com/core-java/java-clone-shallow-copy-and-deep-copy/).
4. ^ "Generic Shallow vs Deep Copy Explanation" (https://secweb.cs.odu.edu/~zeil/cs361/web/website/Lectures/big3/pages/shallowvsdeep.html).
5. ^ "Passing Information to a Method or a Constructor" (http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html). Retrieved 8 October 2013.
6. ^ Java deep-cloning library (http://code.google.com/p/cloning/)

Categories: Object (computer science)
