



WIKIPEDIA  
The Free Encyclopedia

Main page

Contents

Featured content

Current events

Random article

Donate to Wikipedia

Wikipedia store

Interaction

Help

About Wikipedia

Community portal

Recent changes

Contact page

Tools

What links here

Related changes

Upload file

Special pages

Permanent link

Page information

Wikidata item

Cite this page

Print/export

Create a book

Download as PDF

Printable version

Languages

 Add links

Article [Talk](#)

[Read](#)

[Edit](#)

[View history](#)



# Tracing garbage collection

From Wikipedia, the free encyclopedia

(Redirected from [Mark and sweep](#))

In [computer programming](#), **tracing garbage collection** is a form of [automatic memory management](#) that consists of determining which objects should be deallocated ("garbage collected") by tracing which objects are *reachable* by a chain of references from certain "root" objects, and considering the rest as "garbage" and collecting them. Tracing garbage collection is the most common type of [garbage collection](#) – so much so that "garbage collection" often refers to tracing garbage collection, rather than other methods such as [reference counting](#) – and there are a large number of algorithms used in implementation.

## Contents [hide]

- Reachability of an object
- Strong and weak references
- Weak collections
- Basic algorithm
  - Naïve mark-and-sweep
  - Tri-color marking
- Implementation strategies
  - Moving vs. non-moving
  - Copying vs. mark-and-sweep vs. mark-and-don't-sweep
  - Generational GC (ephemeral GC)
  - Stop-the-world vs. incremental vs. concurrent
  - Precise vs. conservative and internal pointers
- Performance
- Determinism
- Real-time garbage collection
- References

## Reachability of an object [\[edit\]](#)

Informally, an object is reachable if it is referenced by at least one variable in the program, either directly or through references from other reachable objects. More precisely, objects can be reachable in only two ways:

- A distinguished set of objects are assumed to be reachable: these are known as the roots. Typically, these include all the objects referenced from anywhere in the [call stack](#) (that is, all [local variables](#) and [parameters](#) in the functions currently being invoked), and any [global variables](#).
- Anything referenced from a reachable object is itself reachable; more formally, reachability is a [transitive closure](#).

The reachability definition of "garbage" is not optimal, insofar as the last time a program uses an object could be long before that object falls out of the environment scope. A distinction is sometimes drawn between [syntactic garbage](#), those objects the program cannot possibly reach, and [semantic garbage](#), those objects the program will in fact never again use. For example:

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();
/* at this point, we know that the Foo object
 * originally assigned to x will never be
 * accessed: it is syntactic garbage
 */

if(x.check_something()) {
    x.do_something(y);
}
System.exit(0);
/* in the above block, y could be semantic garbage,
 * but we won't know until x.check_something() returns
```

```
* some value -- if it returns at all
*/
```

The problem of precisely identifying semantic garbage can easily be shown to be [partially decidable](#): a program that allocates an object  $X$ , runs an arbitrary input program  $P$ , and uses  $X$  if and only if  $P$  finishes would require a semantic garbage collector to solve the [halting problem](#). Although conservative heuristic methods for semantic garbage detection remain an active research area, essentially all practical garbage collectors focus on syntactic garbage.<sup>[[citation needed](#)]</sup>

Another complication with this approach is that, in languages with both [reference types](#) and unboxed [value types](#), the garbage collector needs to somehow be able to distinguish which variables on the stack or fields in an object are regular values and which are references: in memory, an integer and a reference might look alike. The garbage collector then needs to know whether to treat the element as a reference and follow it, or whether it is a primitive value. One common solution is the use of [tagged pointers](#).

## Strong and weak references <sup>[[edit](#)]</sup>

The garbage collector can reclaim only objects that have no references pointing to them either directly or indirectly from the root set. However, some programs require [weak references](#), which should be usable for as long as the object exists but should not prolong its lifetime. In discussions about weak references, ordinary references are sometimes called [strong references](#). An object is eligible for garbage collection if there are no strong (i.e. ordinary) references to it, even though there still might be some weak references to it.

A weak reference is not merely just any pointer to the object that a garbage collector does not care about. The term is usually reserved for a properly managed category of special reference objects which are safe to use even after the object disappears because they *lapse* to a safe value. An unsafe reference that is not known to the garbage collector will simply remain dangling by continuing to refer to the address where the object previously resided. This is not a weak reference.

In some implementations, weak references are divided into subcategories. For example, the [Java Virtual Machine](#) provides three forms of weak references, namely [soft references](#),<sup>[1]</sup> [phantom references](#),<sup>[2]</sup> and regular weak references.<sup>[3]</sup> A softly referenced object is only eligible for reclamation, if the garbage collector decides that the program is low on memory. Unlike a soft reference or a regular weak reference, a phantom reference does not provide access to the object that it references. Instead, a phantom reference is a mechanism that allows the garbage collector to notify the program when the referenced object has become *phantom reachable*. An object is phantom reachable, if it still resides in memory and it is referenced by a phantom reference, but its [finalizer](#) has already executed. Similarly, [Microsoft.NET](#) provides two subcategories of weak references,<sup>[4]</sup> namely long weak references (tracks resurrection) and short weak references.

## Weak collections <sup>[[edit](#)]</sup>

[Data structures](#) can also be devised which have weak tracking features. For instance, weak [hash tables](#) are useful. Like a regular hash table, a weak hash table maintains an association between pairs of objects, where each pair is understood to be a key and value. However, the hash table does not actually maintain a strong reference on these objects. A special behavior takes place when either the key or value or both become garbage: the hash table entry is spontaneously deleted. There exist further refinements such as hash tables which have only weak keys (value references are ordinary, strong references) or only weak values (key references are strong).

Weak hash tables are important for maintaining associations between objects, such that the objects engaged in the association can still become garbage if nothing in the program refers to them any longer (other than the associating hash table).

The use of a regular hash table for such a purpose could lead to a "logical memory leak": the accumulation of reachable data which the program does not need and will not use.

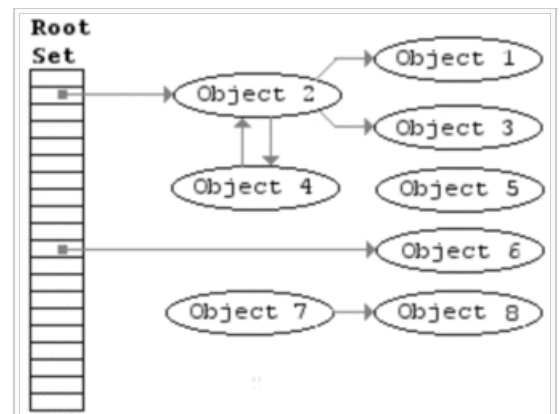
## Basic algorithm <sup>[[edit](#)]</sup>

Tracing collectors are so called because they trace through the working set of memory. These garbage collectors perform collection in cycles. A cycle is started when the collector decides (or is notified) that it needs to reclaim memory, which happens most often when the system is low on memory<sup>[[citation needed](#)]</sup>. The original method involves a naïve **mark-and-sweep** in which the entire memory set is touched several times.

### Naïve mark-and-sweep <sup>[[edit](#)]</sup>

In the naive mark-and-sweep method, each object in memory has a flag (typically a single bit) reserved for garbage collection use only. This flag is always *cleared*, except during the collection cycle. The first stage of collection does a tree traversal of the entire 'root set', marking each object that is pointed to as being 'in-use'. All objects that those objects point to, and so on, are marked as well, so that every object that is ultimately pointed to from the root set is marked. Finally, all memory is scanned from start to finish, examining all free or used blocks; those with the in-use flag still cleared are not reachable by any program or data, and their memory is freed. (For objects which are marked in-use, the in-use flag is cleared again, preparing for the next cycle.)

This method has several disadvantages, the most notable being that the entire system must be suspended during collection; no mutation of the working set can be allowed. This will cause programs to 'freeze' periodically (and generally unpredictably), making real-time and time-critical applications impossible. In addition, the entire working memory must be examined, much of it twice, potentially causing problems in [paged memory](#) systems.



Naive mark-and-sweep in action on a [heap](#) containing eight [objects](#). Arrows represent [object references](#). Circles represent the objects themselves. Objects #1, #2, #3, #4, and #6 are strongly referenced from the root set. On the other hand, objects #5, #7, and #8 are not strongly referenced either directly or indirectly from the root set; therefore, they are garbage.

### Tri-color marking [\[edit\]](#)

Because of these pitfalls, most modern tracing garbage collectors implement some variant of the *tri-color marking abstraction*, but simple collectors (such as the *mark-and-sweep* collector) often do not make this abstraction explicit. Tri-color marking works as described below.

Three sets are created – *white*, *black* and *gray*:

- The white set, or *condemned set*, is the set of objects that are candidates for having their memory recycled.
- The black set is the set of objects that can be shown to have no outgoing references to objects in the white set, and to be reachable from the root. Objects in the black set are not candidates for recycling; in many implementations, the black set starts as empty.
- The gray set contains all objects reachable from the root but yet to be scanned for references to "white" objects. Since they are known to be reachable from the root, they cannot be garbage-collected and will end up in the black set after being scanned. The gray set is initialised to the set of objects which are directly referenced at the root level; typically all other objects are initially placed in the white set.

These three sets [partition](#) the memory; every object in the system, including the root set, is in precisely one set. The algorithm then executes the following:

1. Pick an object from the gray set.
2. "Blacken" this object (move it to the black set) by *graying* all the white objects it references. This confirms that neither this object nor any object it references can be garbage-collected.
3. Repeat the former until the gray set is empty.
4. When there are no more objects in the gray set, the scan has finished; the "black" objects have been shown to be reachable from the root, while the "white" objects have been shown to be unreachable and can be garbage-collected.

Since all objects not immediately reachable from the root are typically assigned to the white set, and objects can only move from white to gray and from gray to black, the algorithm preserves an important invariant – no black object points directly to a white object. This ensures that the white objects can be safely destroyed once the gray set is empty. (Some variations on the algorithm do not preserve the tri-color invariant but use a modified form for which all the important properties hold.)

The tri-color method has an important advantage – it can be performed "on-the-fly", without halting the system for significant periods of time. This is accomplished by marking objects as they are allocated and during mutation, maintaining the various sets. By monitoring the size of the sets, the system can perform garbage collection periodically, rather than as needed. Also, the need to touch the entire working set on each cycle is avoided.

## Implementation strategies [\[edit\]](#)

In order to implement the basic tricolor algorithm, several important design decisions must be made, which can significantly affect the performance characteristics of the garbage collector.

### Moving vs. non-moving [\[edit\]](#)

Once the unreachable set has been determined, the garbage collector may simply release the [unreachable objects](#) and leave everything else as it is, or it may copy some or all of the reachable objects into a new area of memory, updating all references to those objects as needed. These are called "non-moving" and "moving" (or, alternatively, "non-compacting" and "compacting") garbage collectors, respectively.

At first, a moving GC strategy may seem inefficient and costly compared to the non-moving approach, since much more work would appear to be required on each cycle. In fact, however, the moving GC strategy leads to several performance advantages, both during the garbage collection cycle itself and during actual program execution:

- No additional work is required to reclaim the space freed by dead objects; the entire region of memory from which reachable objects were moved can be considered free space. In contrast, a non-moving GC must visit each unreachable object and somehow record that the memory it alone occupied is available.
- Similarly, new objects can be allocated very quickly. Since large contiguous regions of memory are usually made available by the moving GC strategy, new objects can be allocated by simply incrementing a 'free memory' pointer. A non-moving strategy may, after some time, lead to a heavily [fragmented](#) heap, requiring expensive consultation of "free lists" of small available blocks of memory in order to allocate new objects.
- If an appropriate traversal order is used (such as cdr-first for list [conses](#)), objects that refer to each other frequently can be moved very close to each other in memory, increasing the likelihood that they will be located in the same [cache line](#) or [virtual memory](#) page. This can significantly speed up access to these objects through these references.

One disadvantage of a moving garbage collector is that it only allows access through references that are managed by the garbage collected environment, and does not allow [pointer arithmetic](#). This is because any native pointers to objects will be invalidated when the garbage collector moves the object (they become [dangling pointers](#)). For [interoperability](#) with native code, the garbage collector must copy the object contents to a location outside of the garbage collected region of memory. An alternative approach is to **pin** the object in memory, preventing the garbage collector from moving it and allowing the memory to be directly shared with native pointers (and possibly allowing pointer arithmetic).<sup>[5]</sup>

### Copying vs. mark-and-sweep vs. mark-and-don't-sweep [\[edit\]](#)

To further refine the distinction, tracing collectors can also be divided by considering how the three sets of objects (white, grey, and black) are maintained during a collection cycle.

The most straightforward approach is the **semi-space collector**, which dates to 1969. In this moving GC scheme, memory is partitioned into a "from space" and "to space". Initially, objects are allocated into "to space" until they become full and a collection is triggered. At the start of a collection, the "to space" becomes the "from space", and vice versa. The objects reachable from the root set are copied from the "from space" to the "to space". These objects are scanned in turn, and all objects that they point to are copied into "to space", until all reachable objects have been copied into "to space". Once the program continues execution, new objects are once again allocated in the "to space" until it is once again full and the process is repeated. This approach has the advantage of conceptual simplicity (the three object color sets are implicitly constructed during the copying process), but the disadvantage that a (possibly) very large contiguous region of free memory is necessarily required on every collection cycle. This technique is also known as **stop-and-copy**. [Cheney's algorithm](#) is an improvement on the semi-space collector.

A **mark and sweep** garbage collector maintains a bit (or two) with each object to record whether it is white or black; the grey set is either maintained as a separate list (such as the process stack) or using another bit. As the reference tree is traversed during a collection cycle (the "mark" phase), these bits are manipulated by the collector to reflect the current state. A final "sweep" of the memory areas then frees white objects. The mark and sweep strategy has the advantage that, once the unreachable set is determined, either a moving or non-moving collection strategy can be pursued; this choice of strategy can even be made at runtime, as available memory permits. It has the disadvantage of "bloating" objects by a small amount.

A **mark and don't sweep** garbage collector, like the mark-and-sweep, maintains a bit with each object to record whether it is white or black; the gray set is either maintained as a separate list (such as the process stack) or using another bit. There are two key differences here. First, black and white mean different things

than they do in the mark and sweep collector. In a "mark and don't sweep" system, all reachable objects are always black. An object is marked black at the time it is allocated, and it will stay black even if it becomes unreachable. A white object is unused memory and may be allocated. Second, the interpretation of the black/white bit can change. Initially, the black/white bit may have the sense of (0=white, 1=black). If an allocation operation ever fails to find any available (white) memory, that means all objects are marked used (black). The sense of the black/white bit is then inverted (for example, 0=black, 1=white). Everything becomes white. This momentarily breaks the invariant that reachable objects are black, but a full marking phase follows immediately, to mark them black again. Once this is done, all unreachable memory is white. No "sweep" phase is necessary.

## Generational GC (ephemeral GC) [\[edit\]](#)

It has been empirically observed that in many programs, the most recently created objects are also those most likely to become unreachable quickly (known as *infant mortality* or the *generational hypothesis*). A generational GC (also known as ephemeral GC) divides objects into generations and, on most cycles, will place only the objects of a subset of generations into the initial white (condemned) set. Furthermore, the runtime system maintains knowledge of when references cross generations by observing the creation and overwriting of references. When the garbage collector runs, it may be able to use this knowledge to prove that some objects in the initial white set are unreachable without having to traverse the entire reference tree. If the generational hypothesis holds, this results in much faster collection cycles while still reclaiming most unreachable objects.

In order to implement this concept, many generational garbage collectors use separate memory regions for different ages of objects. When a region becomes full, the objects in it are traced, using the references from the older generation(s) as roots. This usually results in most objects in the generation being collected (by the hypothesis), leaving it to be used to allocate new objects. When a collection doesn't collect many objects (the hypothesis doesn't hold, for example because the program has computed a large collection of new objects it does want to retain), some or all of the surviving objects that are referenced from older memory regions are promoted to the next highest region, and the entire region can then be overwritten with fresh objects. This technique permits very fast incremental garbage collection, since the garbage collection of only one region at a time is all that is typically required.

[Ungar's](#) classic generation scavenger has two generations. It divides the youngest generation, called "new space", into a large "eden" in which new objects are created and two smaller "survivor spaces", past survivor space and future survivor space. The objects in the older generation that may reference objects in new space are kept in a "remembered set". On each scavenge the objects in new space are traced from the roots in the remembered set and copied to future survivor space. If future survivor space fills up, the objects that don't fit are promoted to old space, a process called "tenuring". At the end of the scavenge some objects reside in future survivor space and eden and past survivor space are empty. Then future survivor space and past survivor space are exchanged and the program continues, allocating objects in eden. In [Ungar's](#) original system eden is 5 times larger than each survivor space.

Generational garbage collection is a [heuristic](#) approach, and some unreachable objects may not be reclaimed on each cycle. It may therefore occasionally be necessary to perform a full mark and sweep or copying garbage collection to reclaim all available space. In fact, runtime systems for modern programming languages (such as [Java](#) and the [.NET Framework](#)) usually use some hybrid of the various strategies that have been described thus far; for example, most collection cycles might look only at a few generations, while occasionally a mark-and-sweep is performed, and even more rarely a full copying is performed to combat fragmentation. The terms "minor cycle" and "major cycle" are sometimes used to describe these different levels of collector aggression.

## Stop-the-world vs. incremental vs. concurrent [\[edit\]](#)

Simple *stop-the-world* garbage collectors completely halt execution of the program to run a collection cycle, thus guaranteeing that new objects are not allocated and objects do not suddenly become unreachable while the collector is running.

This has the obvious disadvantage that the program can perform no useful work while a collection cycle is running (sometimes called the "embarrassing pause"). Stop-the-world garbage collection is therefore mainly suitable for non-interactive programs. Its advantage is that it is both simpler to implement and faster than incremental garbage collection.

*Incremental* and *concurrent* garbage collectors are designed to reduce this disruption by interleaving their work with activity from the main program. Incremental garbage collectors perform the garbage collection cycle in discrete phases, with program execution permitted between each phase (and sometimes during some phases). Concurrent garbage collectors do not stop program execution at all, except perhaps briefly when the program's execution stack is scanned. However, the sum of the incremental phases takes longer to complete than one



batch garbage collection pass, so these garbage collectors may yield lower total throughput.

Careful design is necessary with these techniques to ensure that the main program does not interfere with the garbage collector and vice versa; for example, when the program needs to allocate a new object, the runtime system may either need to suspend it until the collection cycle is complete, or somehow notify the garbage collector that there exists a new, reachable object.

### Precise vs. conservative and internal pointers [\[edit\]](#)

Some collectors can correctly identify all pointers (references) in an object; these are called *precise* (also *exact* or *accurate*) collectors, the opposite being a *conservative* or *partly conservative* collector. Conservative collectors assume that any bit pattern in memory could be a pointer if, interpreted as a pointer, it would point into an allocated object. Conservative collectors may produce false positives, where unused memory is not released because of improper pointer identification. This is not always a problem in practice unless the program handles a lot of data that could easily be misidentified as a pointer. False positives are generally less problematic on 64-bit systems than on 32-bit systems because the range of valid memory addresses tends to be a tiny fraction of the range of 64-bit values. Thus, an arbitrary 64-bit pattern is unlikely to mimic a valid pointer. A false negative can also happen if pointers are "hidden", for example by the [XOR trick](#). Whether a precise collector is practical usually depends on the type safety properties of the programming language in question. An example for which a conservative garbage collector would be needed is the [C language](#), which allows typed (non-void) pointers to be type cast into untyped (void) pointers, and vice versa.

A related issue concerns *internal pointers*, or pointers to fields within an object. If the semantics of a language allow internal pointers, then there may be many different addresses that can refer to parts of the same object, which complicates determining whether an object is garbage or not. An example for this is the [C++ language](#), in which multiple inheritance can cause pointers to base objects to have different addresses. In a tightly optimized program, the corresponding pointer to the object itself may have been overwritten in its register, so such internal pointers need to be scanned.

## Performance [\[edit\]](#)

Performance of tracing garbage collectors – both latency and throughput – depends significantly on the implementation, workload, and environment. Naive implementations or use in very memory-constrained environments, notably embedded systems, can result in very poor performance compared with other methods, while sophisticated implementations and use in environments with ample memory can result in excellent performance.

In terms of throughput, tracing by its nature requires some implicit runtime [overhead](#), though in some cases the amortized cost can be extremely low, in some cases even lower than one instruction per allocation or collection, outperforming stack allocation.<sup>[6]</sup> Manual memory management requires overhead due to explicit freeing of memory, and reference counting has overhead from incrementing and decrementing reference counts, and checking if the count has overflowed or dropped to zero.

In terms of latency, simple stop-the-world garbage collectors pause program execution for garbage collection, which can happen at arbitrary times and take arbitrarily long, making them unusable for [real-time computing](#), notably embedded systems, and a poor fit for interactive use, or any other situation where low latency is a priority. However, incremental garbage collectors can provide hard real-time guarantees, and on systems with frequent idle time and sufficient free memory, such as personal computers, garbage collection can be scheduled for idle times and have minimal impact on interactive performance. Manual memory management (as in C++) and reference counting have a similar issue of arbitrarily long pauses in case of deallocating a large data structure and all its children, though these only occur at fixed times, not depending on garbage collection.

### Manual heap allocation

- search for best/first-fit block of sufficient size
- free list maintenance

### Garbage collection

- locate reachable objects
- copy reachable objects for moving collectors
- read/write barriers for incremental collectors
- search for best/first-fit block and free list maintenance for non-moving collectors

It is difficult to compare the two cases directly, as their behavior depends on the situation. For example, in the best case for a garbage collecting system, allocation just increments a pointer, but in the best case for manual heap allocation, the allocator maintains freelists of specific sizes and allocation only requires following a pointer.

However, this size segregation usually cause a large degree of external fragmentation, which can have an adverse impact on cache behaviour. Memory allocation in a garbage collected language may be implemented using heap allocation behind the scenes (rather than simply incrementing a pointer), so the performance advantages listed above don't necessarily apply in this case. In some situations, most notably [embedded systems](#), it is possible to avoid both garbage collection and heap management overhead by preallocating pools of memory and using a custom, lightweight scheme for allocation/deallocation.<sup>[7]</sup>

The overhead of write barriers is more likely to be noticeable in an [imperative](#)-style program which frequently writes pointers into existing data structures than in a [functional](#)-style program which constructs data only once and never changes them.

Some advances in garbage collection can be understood as reactions to performance issues. Early collectors were stop-the-world collectors, but the performance of this approach was distracting in interactive applications. Incremental collection avoided this disruption, but at the cost of decreased efficiency due to the need for barriers. Generational collection techniques are used with both stop-the-world and incremental collectors to increase performance; the trade-off is that some garbage is not detected as such for longer than normal.

## Determinism [\[edit\]](#)








- Tracing garbage collection is not [deterministic](#) in the timing of object finalization. An object which becomes eligible for garbage collection will usually be cleaned up eventually, but there is no guarantee when (or even if) that will happen. This is an issue for program correctness when objects are tied to non-memory resources, whose release is an externally visible program behavior, such as closing a network connection, releasing a device or closing a file. One garbage collection technique which provides determinism in this regard is [reference counting](#).
- Garbage collection can have a nondeterministic impact on execution time, by potentially introducing pauses into the execution of a program which are not correlated with the algorithm being processed. Under tracing garbage collection, the request to allocate a new object can sometimes return quickly and at other times trigger a lengthy garbage collection cycle. Under reference counting, whereas allocation of objects is usually fast, decrementing a reference is nondeterministic, since a reference may reach zero, triggering recursion to decrement the reference counts of other objects which that object holds.

## Real-time garbage collection [\[edit\]](#)

While garbage collection is generally nondeterministic, it is possible to use it in hard [real-time](#) systems. A real-time garbage collector should guarantee that even in the worst case it will dedicate a certain number of computational resources to mutator threads. Constraints imposed on a real-time garbage collector are usually either work based or time based. A time based constraint would look like: within each time window of duration  $T$ , mutator threads should be allowed to run at least for  $Tm$  time. For work based analysis, MMU (minimal mutator utilization)<sup>[8]</sup> is usually used as a real-time constraint for the garbage collection algorithm.

One of the first implementations of [hard real-time](#) garbage collection for the [JVM](#) was based on the Metronome algorithm,<sup>[9]</sup> whose commercial implementation is available as part of the [IBM WebSphere Real Time](#).<sup>[10]</sup> Another hard real-time garbage collection algorithm is Staccato, available in the [IBM's J9 JVM](#), which also provides scalability to large multiprocessor architectures, while bringing various advantages over Metronome and other algorithms which, on the contrary, require specialized hardware.<sup>[11]</sup>

## References [\[edit\]](#)

1. <sup>^</sup> ["Class SoftReference<T>"](#) . *Java™ Platform Standard Ed. 7*. Oracle. Retrieved 25 May 2013.
2. <sup>^</sup> ["Class PhantomReference<T>"](#) . *Java™ Platform Standard Ed. 7*. Oracle. Retrieved 25 May 2013.
3. <sup>^</sup> ["Class WeakReference<T>"](#) . *Java™ Platform Standard Ed. 7*. Oracle. Retrieved 25 May 2013.
4. <sup>^</sup> ["Weak References"](#) . *.NET Framework 4.5*. Microsoft. Retrieved 25 May 2013.
5. <sup>^</sup> ["Copying and Pinning"](#) . Msdn2.microsoft.com. Retrieved 9 July 2010.
6. <sup>^</sup> Appel, Andrew W. (17 June 1987). "Garbage collection can be faster than stack allocation" . *Information Processing Letters* **25** (4): 275–279. doi:10.1016/0020-0190(87)90175-X.
7. <sup>^</sup> ["Memory allocation in embedded systems"](#) . Eros-os.org. Retrieved 29 March 2009.
8. <sup>^</sup> Cheng, Perry; Blelloch, Guy E. (22 June 2001). "A parallel, real-time garbage collector". *ACM SIGPLAN Notices* **36** (5): 125–136. doi:10.1145/381694.378823.
9. <sup>^</sup> ["The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems"](#)  (PDF).
10. <sup>^</sup> ["Real-time Java, Part 4: Real-time garbage collection"](#) .
11. <sup>^</sup> McCloskey, Bill; Bacon, David F.; Cheng, Perry; Grove, David (22 February 2008). "Staccato: A Parallel and Concurrent Real-time Compacting Garbage Collector for Multiprocessors"  (PDF). Retrieved 11 March 2014.







Categories: [Automatic memory management](#)

This page was last modified on 29 August 2015, at 19:40.

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.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

