Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

# Προηγμένες Μέθοδοι Προγραμματισμού

ΠΜΣ (M135.CS1E, M135.CS23B, M135.IC1E, παλαιό: M117)

## Garbage Collection

Δρ. Κώστας Σαΐδης (saiko@di.uoa.gr)

# Manual memory management

The programmer has to:

- Allocate the memory required to hold the contents of the data structures.
- Remember to deallocate (free) it, in order to avoid memory leaks.

# Automatic memory management

The programmer has nothing to do :)

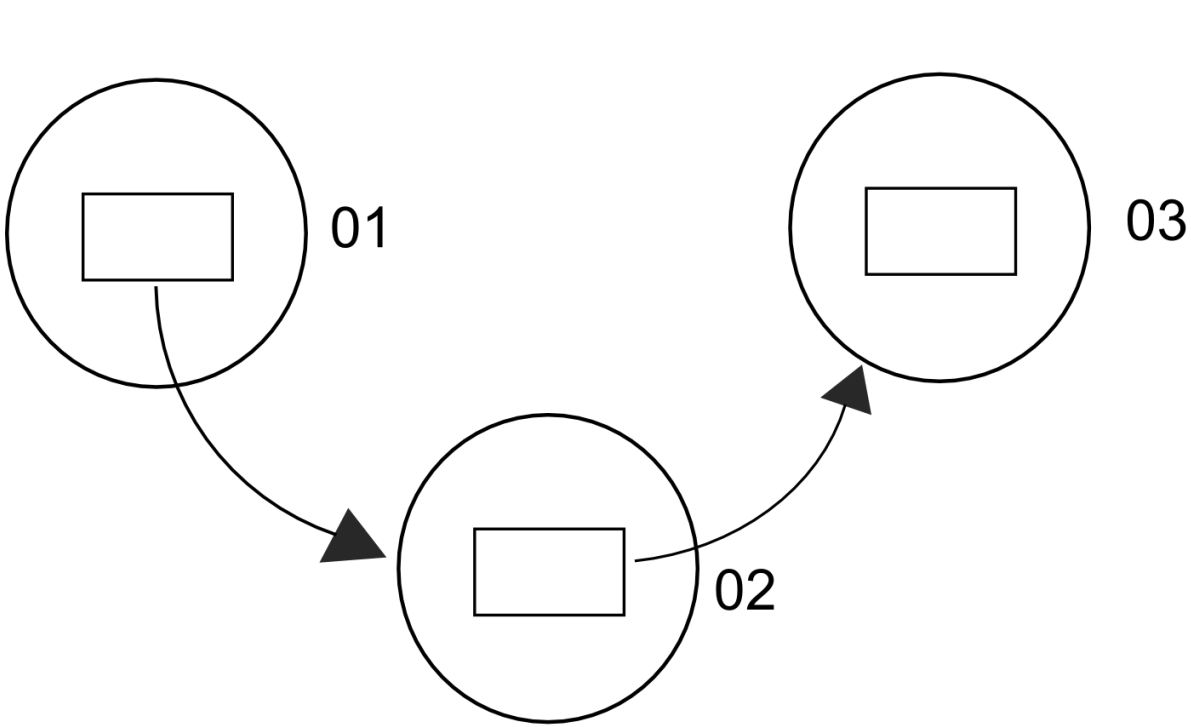
The runtime environment employs a garbage collection mechanism (the garbage collector) that takes care of allocating/de-allocating memory transparently.

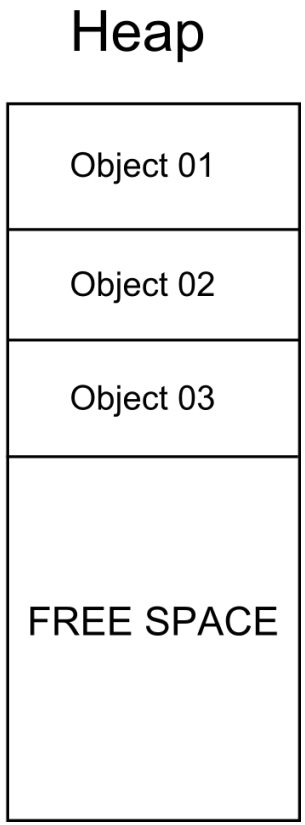
# Benefits

1. The code is easier to write, read, understand and maintain.
2. Increases the level of abstraction and advances productivity.

# The heap

- Objects reside in the heap.
- Objects reference each other.



# The garbage collector

1. Determines whether an object is referenced.
2. If not, it reclaims the memory used by the object and delivers it back to the executing program.

# Desired attributes

The garbage collector should be:

- Safe: never reclaim a "live" object.
- Concise: never allow a dead object to be kept for more than a small number of collections
- Efficient: does not take too much time to finish (the execution pauses should be as short as possible)
- Scalable: memory allocation/deallocation scales well under heavy load in multithreaded systems

# Classification of garbage collectors (I)

Based on the algorithmic approach

- Reference counting:
  - Maintain a reference counter per object
  - Cannot handle ref cycles.
- Reference tracing (Mark & sweep)
  - Mark phase: traverse the reference graph and mark unreachable objects
  - Sweep phase: free memory

# Reachable objects

- Define a set of roots (e.g. main class, static methods/variables, etc. depends on the implementation)
- An object is reachable by the executing program if there is a path of references from the roots that accesses the object.

# Classification (II)

Based on the parallelism of the collector implementation

- Serial: a single collection is executed at a time.
- Parallel: the collection is devised in sub-tasks which are executed in parallel.

# Classification (III)

Based on the program pauses introduced by the collector

- Stop the world: the collection cannot happen concurrently to the execution of the program, the application is being suspended during the collection.
- Concurrent: the collector runs concurrently to the program (yet some pauses may occur).

# Classification (IV)

Based on the handling of heap fragmentation

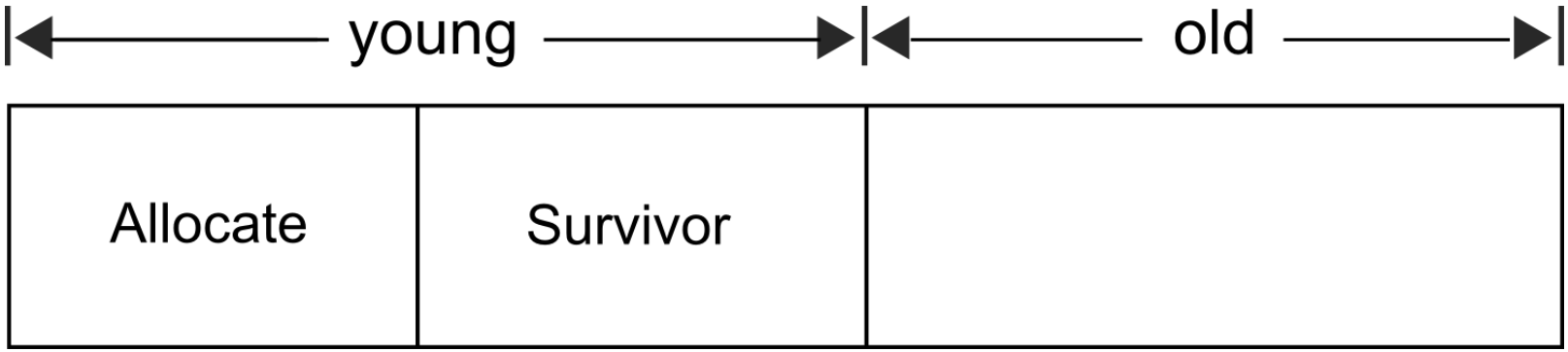
- Compacting: moves and rearranges reachable/live objects in contiguous heap space blocks. The remaining space is free space.
- Noncompacting: leaves the free heap space of the non-reachable objects as-is.
- Copying: copies all reachable objects in a different contiguous area of the heap space. The source area is free area.

# The generational hypothesis

- Most objects are short-lived (they die very quickly).
- Some objects have very long lifetime.
- Old objects usually reference a few younger ones.

# The generational collectors

- Group objects by their "generation" to offer shorter collection pauses:
  - Classify objects based on their generation
  - move them from younger to older generations depending on how many collections the objects manage to survive.



# Shorter collection pauses

- Multiple generations are GCed independently:
  - improves locality.
  - reduces work to be done.
- The garbage collection of the young generation involves mostly dead objects which can be quickly reclaimed
- The collector doesn’t spend time copying and moving the same long-lived objects.
- The younger generations are collected more frequently.

# Criteria for choosing a garbage collector

# Throughput

- The percentage of total time spent in useful application activity versus memory allocation and garbage collection.
- For example, if your throughput is 95%, that means the application code is running 95% of the time and garbage collection is running 5% of the time. You want higher throughput for any high-load business application.

# Latency

- Application responsiveness, which is affected by garbage collection pauses.
- In any application interacting with a human or some active process, you want the lowest possible latency.

# Footprint

- The total memory consumption of the running application (or more accurately, of the whole JVM instance).

# GC Survey

https://yanniss.github.io/M135-21/gcsurvey.pdf

# In practical terms (summing up)

Three steps involved in the garbage collection process:

- Mark
- Sweep
- Compact

We usually want these actions to occur concurrently with our running program, causing the smallest "stop-the-world" pauses possible.

# The main garbage collectors supported by OpenJDK 17

https://docs.oracle.com/en/java/javase/17/gctuning/

- Serial collector
- Parallel collector
- G1
- Z

# The serial collector

- A generational collactor that uses a single thread to perform all garbage collection work.
- It's best-suited to single processor machines.
- It can be useful on multiprocessors for applications with small data sets (up to approximately 100 MB).
- Enabled with the `-XX:+UseSerialGC` option.

# The parallel collector

- Also known as the throughput collector.
- A generational collactor that uses multiple threads to perform the garbage collection work.
- Supports parallel compaction, the major collections are performed in parallel.
- Intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware.
- Enabled with the `–XX:+UseParallelGC` option.

# The G1 (Garbage-first) collector

- A mostly concurrent collector, performing expensive work concurrently to the application.
- Designed to scale from small machines to large multiprocessor machines with a large amount of memory.
- It provides the capability to meet a pause-time goal with high probability, while achieving high throughput.
- G1 is selected by default on most hardware and operating system configurations, or can be explicitly enabled using `-XX:+UseG1GC`.

# The Z collector

- A scalable low-latency garbage collector, that performs all expensive work concurrently, without stopping the execution of application threads.
- Provides max pause times of a few milliseconds, but at the cost of some throughput.
- It is intended for applications, which require low-latency, while pause times are independent of the heap size that is being used.
- It supports heap sizes from 8MB to 16TB.
- Enabled with the `-XX:+UseZGC` option.

# Basic memory & GC monitoring tools

- jinfo
- jstat
- jconsole

# JVM options that affect memory use

-Xms: Sets the minimum and initial size of the heap.
-Xmx: Sets the maximum size of the heap.
-XX:MetaspaceSize: Sets the initial size of Metaspace.
-XX:MaxMetaspaceSize: Sets the maximum size of Metaspace.

Production environments often set the -Xms and -Xmx options to the same value so that the heap size is fixed and pre-allocated to the JVM.

# Calculating JVM memory consumption

```
JVM memory =
  Heap memory +
  Metaspace +
  CodeCache +
  (ThreadStackSize * Number of Threads) +
  DirectByteBuffers +
  JVM-native
```

# JVM memory components

- Heap memory: Stores the objects of the application.
- Metaspace: Stores information about the classes and methods used in the application (class metadata).
- CodeCache: Stores the native code generated by the JVM (due to JIT compilation, mainly).
- DirectByteBuffers: Direct buffer pools (Latest Java/JVM versions allows an app to allocate native off-heap memory).
- JVM-native: JVM internals.

# The life-cycle of objects in the JVM

- Object reachability states
- Object finalizers
- The java.lang.ref package

# Object reachability states

- Reachable: An object begins its life in this state and remains reachable as long as the garbage collector can “reach” it by traversing the reference graph starting from the roots.
- Resurrectable: An object is resurrectable if it is unreachable, yet, potentially, it could be made reachable again by a finalizer (through the finalize() method).
- Unreachable: An object is not reachable and it cannot be made reachable by the execution of a finalizer.

# Object Finalization

- The method `protected void finalize()` of `java.lang.Object`. It is (supposed to be) called by the garbage collector.
- This method is supposed to be overridden in order to perform object specific finalizations.

# However

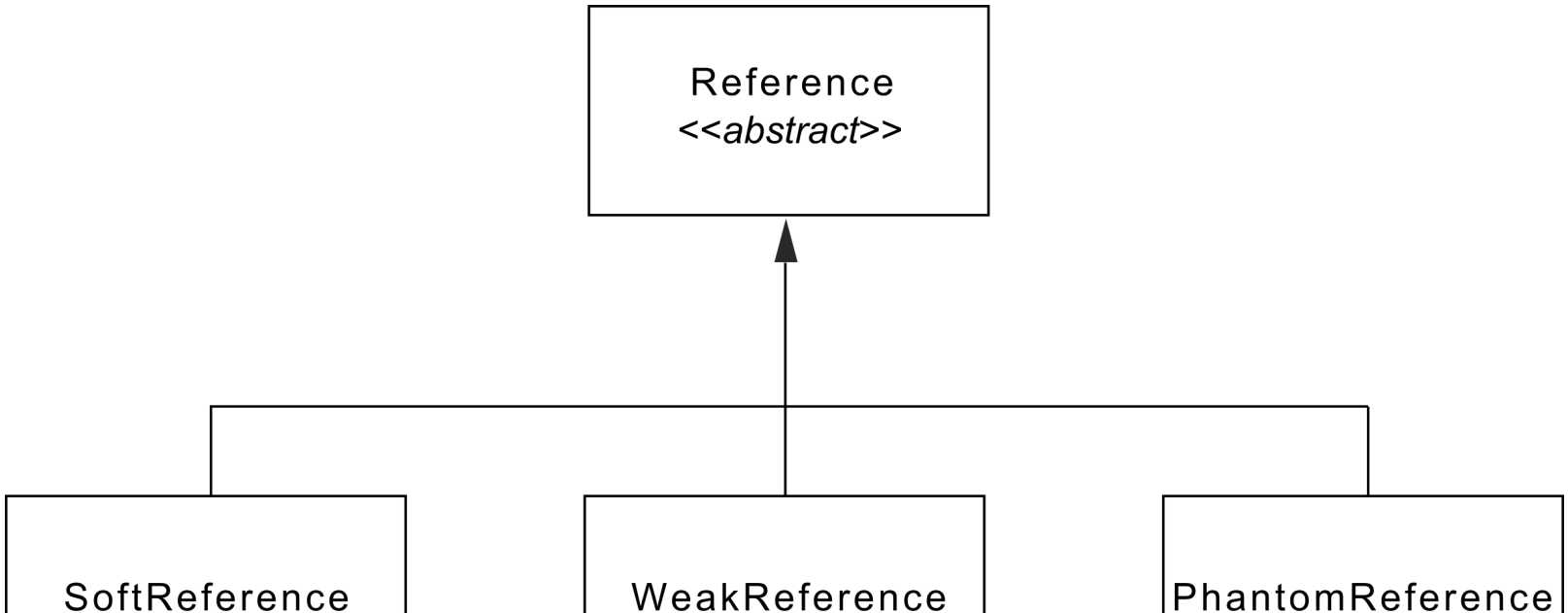
- There is no guarantee that this method will ever be executed.
- If it will be executed, the guarantee is that it will be executed only once.
- Don’t implement your application logic to rely on this method.
- Avoid to override the `finalize()` method. Finalize your objects in another fashion (e.g., by using a shutdown hook or by implementing and calling a dispose() method), always adding an appropriate finally block).

# The java.lang.ref Package

- Finer-grained reachable states - interaction with the garbage collector
- Reachable
  - Strongly reachable. Ordinary object references.
  - Softly reachable (SoftReference). Well-suited for caches.
  - Weakly reachable (WeakReference). Well-suited for canonicalizing mappings.
  - Phantom reachable (PhantomReference). Well-suited for premortem cleanup (instead of finalize()). This is the only way to determine when an object is removed from memory.

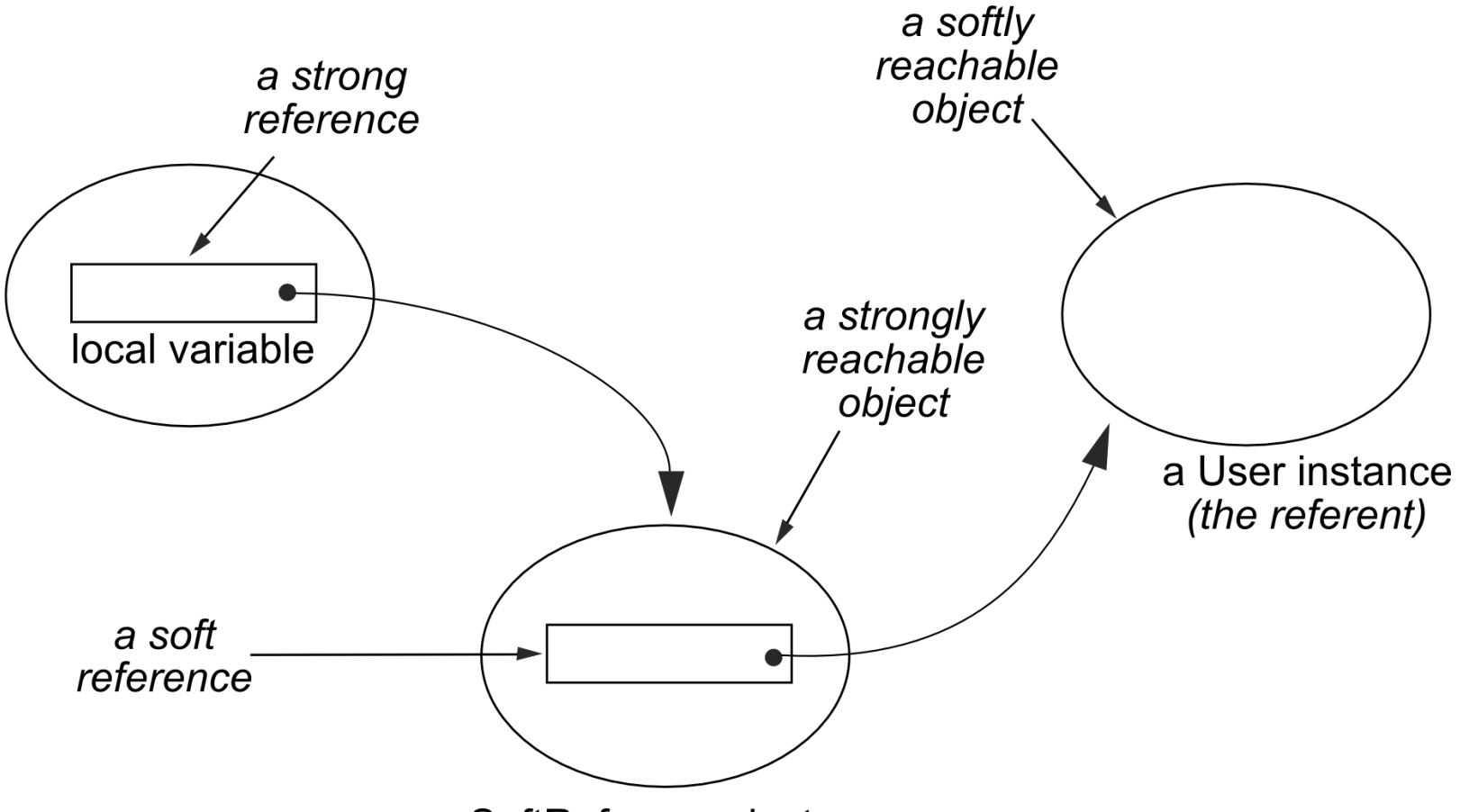
# The Reference Object

- A reference object holds a reference to another object, called the referent.
- Whereas a strong reference prevents its referent from being garbage collected, soft, weak, and phantom references do not!



# Example

```
User user = new User("user");
SoftReference<User> ref = new SoftReference<>(user) ;
```



# Reference object methods

- `get()` : returns the referent.
- `clear()` : invalidates the referent.

```
User user = ref.get() ;
println ("User : " + user ) ;
ref.clear() ;
user = ref.get() ;
println( "User : " + user ) ;
```

```
> User: user
> User: null
```

# The java.lang.ref.ReferenceQueue

- Used by the garbage collector to inform the program about an object's state changes.
- A SoftReference or WeakReference object may be optionally associated with a ReferenceQueue.
- A PhantomReference is always associated with a ReferenceQueue.

# Associating a Reference with a Queue

```
ReferenceQueue q = new ReferenceQueue();
User user = new User("user");
SoftReference<User> ref = new SoftReference<>(user, q) ;
```

# Monitoring the ReferenceQueue

- The collector will append a Reference to the ReferenceQueue when the referent's reachability state changes.
- The application can monitor the ReferenceQueue using:
  - `ReferenceQueue.poll()`: Removes the object waiting in the queue (if any) and returns it. If the queue is empty it immediately returns null (nonblocking).
  - `ReferenceQueue.remove(timeout)`: As above, but blocking. It will block until an object is available in the queue or until the timeout expires.

# Reachability States Revisited

- Strongly reachable: the object can be reached through an ordinary reachable reference. The collector does not attempt to reclaim the memory used by this object.
- Softly reachable: the referent of a strongly reachable SoftReference.
- Weakly reachable: the referent of a strongly reachable WeakReference.
- The collector may attempt to reclaim the memory used by softly and weakly reachable objects, clearing and enqueuing their respective Reference object.

# Reachability States Revisited

- Resurrectable: The object is not strongly, softly or weakly reachable, but it may be resurrected back to its original state (strong, soft, weak) by a finalizer.
- Phantom reachable: The referent of a strongly referenced PhantomReference, after the execution of the referent's finalizers (if any).
- Unreachable: To be reclaimed.

**The get() method of a PhantomReference always returns null.**

# Soft/Weak references and the Queue

- The garbage collector enqueues soft and weak references when their referents are leaving the relevant state.
- When the collector needs to reclaim a soft or weak reference, it first clears the referent of the respective reference object.
- Then, either immediately or at a later point in time, it adds the cleared reference to its associated queue (if such a queue is present).

# Phantom references and the Queue

The garbage collector enqueues phantom references when the referents are entering the relevant state (they have been finalized).

- That is, the garbage collector enqueues PhantomReference objects to indicate that their referents have entered the phantom reachable state.
- Phantom reachable objects will remain phantom reachable until their reference objects are explicitly cleared by the program (by calling the
  clear() method).

# When to use Weak References

- When you need to "hold" some ephemeral data/state that you don't need it to survive a collection (the JVM collects weak references eagerly).
- Mainly used for implementing canonicalizing mappings.
- A weak reference isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.

# When to use Soft References

- When you need to "hold" some ephemeral data/state that you need it to survive as many collections as possible (the JVM collects soft references less eagerly, usually as part of its effort to prevent/avoid memory exhaustion).
- Mainly used for implementing memory-resident caches.
- A soft reference is exactly like a weak reference, except that it is less eager to throw away the object to which it refers. An object which is only weakly reachable (the strongest references to it are WeakReferences) will be discarded at the next garbage collection cycle, but an object which is softly reachable will generally stick around for a while.

# When to use Phantom References

- Mainly for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.
- They are not automatically cleared by the garbage collector as they are enqueued. An object that is reachable via phantom references will remain so until all such references are cleared or themselves become unreachable.

# Example

Let's see the `GCTest*` examples on the course's repo.

Check out how the `org.apache.commons.io.FileCleaningTracker` class, which uses phantom references and the reference queue to delete files when marker objects enter the phantom-reachable state.

# java.util.WeakHashMap

- Well-suited for canonicalizing mappings.
- It contains weak keys (not weak values).
- Not to be used for caching!

# Using SoftReference for Caching

```
public class SoftReferenceCache<K, V> {

  private final Map<K, SoftReference <V>> map = new HashMap<>();

  public void put (K key, V value) {
    map. put (key, new SoftReference<V>(value));
  }

  public V get (K key) {
    SoftReference <V> reference = map.get(key) ;
    if (reference == null) { //the key is not in the cache
          return null;
    }
    else { //the referent may be null (GCed), users may not expect this
      return reference.get();
    }
  }
}
```

# Memory Leakage

- The SoftReference objects remain strongly referenced within the map, yielding softly-reachable referents.
- Modify the get() method to remove the SoftReference objects from the map when their referrents are null.

```
public V get(K key) {
  SoftReference<V> reference = map.get(key) ;
  if ( reference == null ) { //the key is not in the cache
    return null ;
  } else {
    V v = reference.get() ;
    if(v == null) map.remove(key);
    return v ;
  }
}
```