Garbage Collection in Java

see web pages for links
Two extremes

- Programmer not responsible at all for memory management
- Programmer has to be very careful to please GC

but:

- need to understand GC for large-scale programs
- (potentially) huge impact on speed
GC guarantees

- Java Language specification (JLS) rather vague
- Java Virtual Machine Spec (JVMS):
  - heap is created at JVM startup
  - heap storage for objects is automatically reclaimed (GC); no explicit deallocation
  - no specific mechanism specified
  - why?
Object Lifecycle

- Creation
- In use (strongly reachable)
- Invisible
- Unreachable
- Collected
- Finalized
- Deallocated
Creation

- allocate space
- begin object construction
- call superclass constructors (recursively)
- run instance initializers and instance variable initializers
- execute the body of the constructor
In use

- in use: held by at least one strong reference
- all references are strong (unless we explicitly use one of: soft, weak, or phantom refs, see later)

```java
public class CatTest {
    static Vector catList = new Vector();
    static void makeCat() {
        Object cat = new Cat();
        catList.addElement(cat);
    }
    public static void main(String[] arg) {
        makeCat();
        // do more
    }
}
```
Invisible

- no strong reference left that is accessible to the program, but there are still references: local vars that have gone out of scope:

```java
public void run() {
    try {
        Object foo = new Object();
        foo.doSomething();
    } catch (Exception e) {
        // whatever
    }
    while (true) { // ... loops forever
```
Unreachable

• no more direct or indirect strong references from the rootset to an object ==> candidate for collection

• rootset:
  – temp vars on the stack (all threads)
  – static vars (all loaded classes)
  – all references coming from native code (JNI)

• Note: circular refs are no problem with this def
Example

public void buildDog() {
    Dog newDog = new Dog();
    Tail newTail = new Tail();
    newDog.tail = newTail;
    newTail.dog = newDog;
}

Collected, Finalized, Deallocated

• if the collector finds an object unreachable:
  – if it has a “finalize” method: mark for finalization (i.e. final deallocation will be delayed)

• finalizers have been run ==> finalized
  – no guarantees about “when”, may actually not even run before termination
  – finalizers are rarely a good idea
  – beware of “resurrection”

• deallocation: reuse space (again, whenever)
Reference objects

- java.lang.ref to help tune GC, prevent leaks
- soft, weak, and phantom refs, reference queues
  - soft refs for implementing memory-sensitivities caches
  - weak refs for mappings where keys or values may be reclaimed
  - (phantom refs for final cleanup, better than finalizers)
Reachability

• strongly reachable: only strong refs from rootset
• softly: not strongly, but through live soft ref
  – will only collected (+ set ref to null) if space is scarce
• weakly: neither strong nor soft, but live weak ref
  – will be collected and ref is set to null
• (phantom: neither strong, weak, nor soft, and has been finalized)
• else: unreachable, ==> reclaim space
Example

Thread active

Thread finished

after GC