Effective Java Programming

memory management

Structure

- memory management
  - memory management
  - how garbage collection works
  - types of references
  - how memory leaks occur in Java
  - bad practices – what to avoid
  - reducing memory usage
  - fine tuning the garbage collector

Motto

"Everything should be made as simple as possible, but not simpler."

Albert Einstein

Memory management

- there is no open memory management in Java
  - new objects can be created
  - there is no way to free memory after unused objects
- memory is managed by a separate *Garbage Collector* threat
  - locates and removes objects, which do not have any connection to active threads
  - locates and removes islands of objects
- *GC* cannot be forced to clear memory
  - *System.gc()* – only suggests to clear memory, can be ignored
  - *Runtime.getRuntime().gc()* – same as above

How garbage collection work

- *GC* is the mostly misunderstood feature of Java
  - some think, that it is solely responsible for memory management
  - others try to help the GC, resulting in more work than necessary
- understanding GC mechanisms is crucial for highly efficient, robust software

What guarantees GC?

- JVM specs give little promises on how GC actually works
  - heap is created when JVM starts
  - heap is managed by GC
  - objects are never directly cleared
  - JVM does not specify any memory management algorithm
  - memory management algorithm can be selected according to system requirements
- although every JVM can have a different memory management algorithm, all share the same object's life cycle model

Object's life cycle

- created
- in use
- invisible
- unreachable
- collected
- finalized
- deallocated

Object's life cycle (created)

- when an object is created:
  - heap memory gets allocated
  - object creation gets started
  - constructor of super class gets invoked
  - object's attributes get initialized
  - rest of the constructor is run
- this results in a pitfall
  - NEVER call other methods from a constructor
- object's creation cost depends on JVM implementation, but always exists
- after creation, the object goes into the *in use* state

Object's life cycle (in use)

- objects accessible through at least one strong reference are in the in use state
- in Java 1.1 there where only strong references
- later new types of references have been introduced
  - soft
  - weak
  - phantom

Object’s life cycle (in use)

- after adding element to the list we have two strong references to *Cat*

```
public class Test {
  static List list = new ArrayList();
  static void makeCat() {
    Object cat = new Cat();
    list.add(cat);
  }
  public static void main (String ... args) {
    makeCat();
  }
}
```

Object's life cycle (invisible)

- object is in *invisible* state if there are no strong references **available to the program**, although references may exist
- not all objects go through this state
  - normally, when there are no strong references, object becomes *unreachable*
  - for better performance, JVM may wait till the end of the method before removing references from stack

Object's life cycle (invisible)

```
public void run() {
  try {
    Cat cat = new Cat();
    cat.doSomething();
  } catch(Exception e) { ... }
  while (true) { ... }
}
```

- after leaving the *try* block, there are no references to the *cat* object
- it seems, the object is unreachable
  - no code can access this object
- most efficient JVM implementations do not delete the reference after leaving scope
  - object has a strong reference at least till the end of *run* method
  - in this case – *cat* is invisible for a long time, but is not unreachable, cannot be cleared!

Object’s life cycle (unreachable)

- object is unreachable, if there are no strong references available to the threat
  - GC locates and clears islands of objects
- object in this state is a **candidate** for clearing
  - it won’t be cleared immediately
  - JVM can postpone collecting until necessary
- cyclic references do not always lead to memory leaks

Islands of objects

```
void BuildCar() {
  Car car = new Car();
  Tire tire = new Tire();
  car.tire = tire;
  tire.car = car;
} // before exiting

void testCarBuilding() {
  buildCar();
} // after exiting
```

Object's life cycle (collected)

- object is marked as *collected* if was recognized as unreachable by GC and is being prepared for collecting
  - if it has *finalize* method, will be marked as for finalization
  - otherwise, will be marked as *finalized*
- *finalize* method (if present)
  - has to be called on every object before collecting
  - garbage collection is delayed by this method
  - many objects can await for calling *finalize*, still being present in memory

finalize method

- shouldn't be widely used!
- *finalize*
  - delays creating objects
    - JVM has to mark the object as finalizable
  - extends lifecycle (delays clearing)
    - important for short living objects (there are the majority)
  - can increase the size of the object
    - some JVM add a special attribute to keep the object in a special queue

finalize method

- *finalize*
  - should only be used to deallocate resources not managed by JVM
    - memory used by native code
    - files, sockets, db connections
  - there is no guarantee, the method will be called
    - object can never be collected
      - program ends and all memory is returned to OS
    - common mistake is to place substantial logic there
      - **it is not a destructor as known from C++**

Object's life cycle (finalized)

- object is in this state if still unreachable after calling *finalize*
  - method can "resurrect" object by tying it to a static variable
    - bad idea – finalize won't be called again
    - leads to serious problems
  - *finalized* can be called only once
  - won't be called again if object will again be in *collected* state
- in this state the objects waits to be cleared

Object's life cycle (deallocated)

- last state in *GC* model
- object has been cleared
- previously used space can be allocated
- there is no way to recall a cleared object

Types of references

- java.lang.ref contains classes allowing better cooperation with GC
- defines 3 new types of references (inherit after *Reference*)
  - SoftReference
  - WeakReference
  - PhantomReference
- every type defines a different type of garbage collection when object is only reachable through *Reference*
  - *Reference* holds a reference to an object
    - it behaves like a proxy
      - *get()* returns the object
    - additionally allows garbage collection when nothing uses the object

SoftReference

- strongest reference (from *Reference*)
- used to implement memory-sensitive cache
- can be decided, whether or not to place object in *ReferenceQueue*
  - if reference has one assigned, GC will place it there after removing physical object
  - we can get reference from queue and clean it up

SoftReference

- a *softly reachable* has only soft references, no strong ones
- after locating softly reachable objects, the GC:
  - decides, if it should clear them
    - it is guaranteed they will be cleared before *OutOfMemoryError*
    - no other assumptions for garbage collection
  - at the same time or later cleared references will be enqued

WeakReference

- "weaker" than SoftReference
- designated for map implementation, which keys and values can be cleared (*WeakHashMap*)
- can be decided whether or not to put the object in *ReferenceQueue*
- a weakly reachable object has no strong or soft references
- after locating weakly reachable object, the GC
  - clears the references
  - declares weakly reachable object for finalization
  - at the same time or later queue weak references

PhantomReference

- weakest reference
- allows performing actions before object's destruction in more elastic way than *finalize* method
- object has to be placed in *ReferenceQueue*
- a phantom reachable object has no strong, soft or weak references
- after locating phantom reachable objects
  - GC queues them (when found or later)
  - they are not automatically cleared
  - should be cleared manually or left
- *get()* always returns null
  - cannot get a wrapped object
  - lost objects stay that way

How memory leaks occur in Java

- only strong references lead to memory leaks on heap
  - forgotten references to unused objects
    - main reason is that the reference will be overridden by a new one by next use, which can never occur
  - “lost” objects in collections
    - unused elements in sets
    - values in maps under unused keys
    - objects in collections with hash tables, in which the *hashCode()* value changed
    - remember WeakHashMap<K,V>

How memory leaks occur in Java

- soft and weak references don't lead to leaks
- some sources say, that phantom references also, but…
  - they are not automatically cleared
  - if a reference taken from queue is not cleared or dropped, a leak occurs

How memory leaks occur in Java

- *finalize* can lead to leaks
  - objects with overridden finalize method, marked by GC as unused, are send to finalization queue
    - this queue can get very big
    - no object will be cleared before finalizing
    - there is no time guarantee
    - delays clearing
  - objects “resurrected” in *finalize*
    - always a bad idea
    - if the whole world forgot them, resurrecting often leads to leaks

How memory leaks occur in Java

- exceptions can change the flow of control
  - cleaning code can be skipped
    - elements stay in collections
    - event listeners
  - ALWAYS put this code in finally
- objects of inner classes keep reference to outer class
  - when outer class not used, define inner class as static

How memory leaks occur in Java

- some solutions allow keeping objects in scopes
- servlets define 4 scopes
  - application – whole application life time
  - session – from first call till end of session
    - *invalidate()*
    - *time-out* – can be indefinite and user does not log out
  - request – from call till response
  - page – from beginning generating view till end
- common problem – session “sweeling”
  - session keep data required for one request
  - is not cleared afterwards
- ALWAYS define objects in lowest possible scope

How memory leaks occur in Java

- can happen through stack
  - references in *invisible* state can not be cleared immediately
  - JVM delays it till removing from heap – end of method
  - if leaving a block object gets invisible, its life time is extended till end of method
  - method can do time expensive calculations (*invisible* example)
  - reference *null*

How memory leaks occur in Java

- for previous case
  - carefully reference *null*
  - profiling should show if it is necessary
  - JIT can do it for you

```
void example() {
  int[] array = new int[1024];
  fill(array);
  show(array); // last use
  array = null; // NOT NECESARRY
  // GC sees, that array is not used
}
```

How memory leaks occur in Java

- memory leaks can also not apply to memory managed by JVM
- native methods (JNI) use code written in other languages
  - memory from outside JVM stack gets allocated
  - hard to define memory used by application
  - GC cannot manage this
  - lack of managing code leads to leaks
  - manually clean this memory
    - *finalize* for object that uses JNI
    - *PhantomReference*

Bad practices – what to avoid

- creating big objects
  - longer allocation time
  - longer initialization time
    - more attributes to set to default (null or 0)
  - often to big for cache (Eden space)
  - allocated in space for older objects
    - seldom cleaning
  - can cause memory fragmentation during clear
- **GC loves small objects**
  - easy to allocate memory
  - optimization mechanisms
    - placed in Eden (cache)
    - optimized lists for popular allocation sizes

Bad practices – what to avoid

- System.gc() – application has too little information
  - never periodically
  - bad timing – hurts efficiency
  - occasionally
    - MAYBE in clearly defined places
    - when efficiency is not important (night)
- let the GC work
  - –XX:+DisableExplicitGC

Reducing memory usage

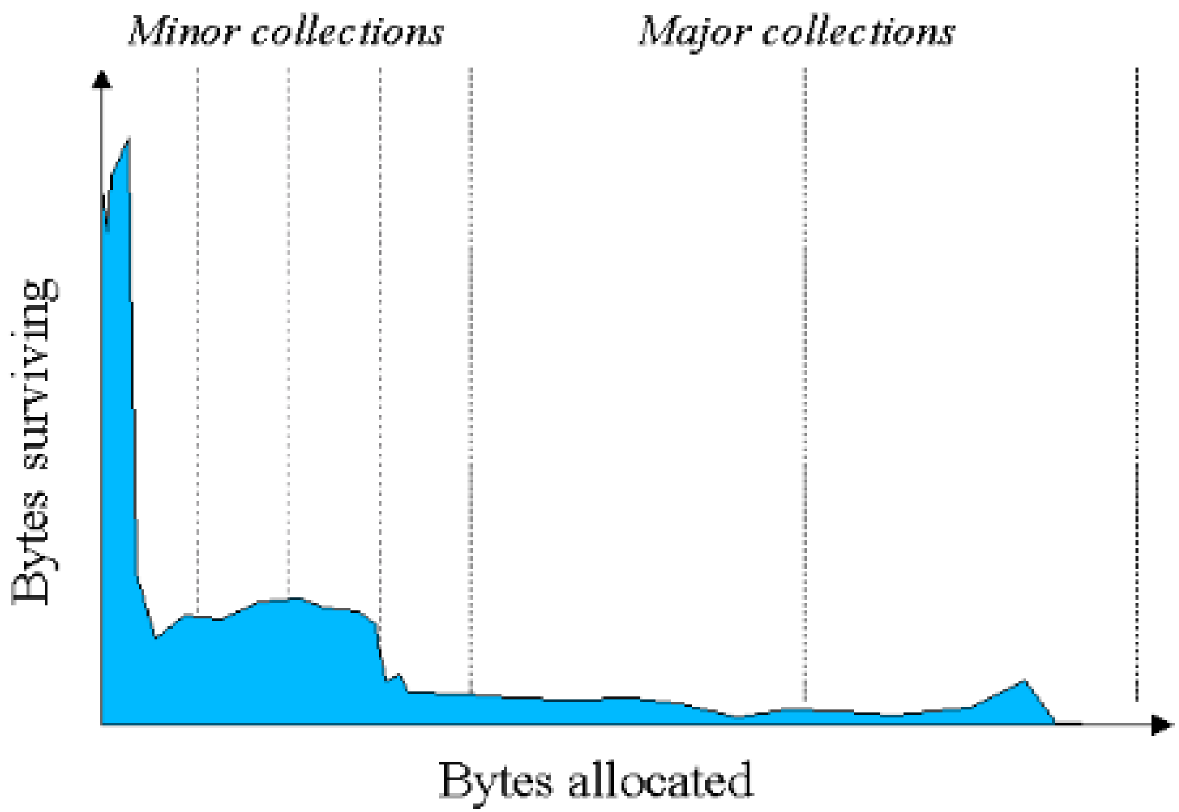
- avoid memory leaks :)
- remember how to reduce number of classes
  - use sparingly
  - reflections use more CPU time
- remember about soft reference collections
- avoid overriding *finalize*
- use lowest possible scope
- tune collections to needed size

Other GC tuning options

- Java has many configuration parameters for GC
  - parameters starting with -X
    - custom
    - not guaranteed to work in all JVM
    - can be changed in later versions
  - parameters starting with -XX
    - unstable
    - not recommended for everyday use
    - can be changed in later versions

Weak generational hypothesis

- most of the objects die young



Weak generational hypothesis

- to optimize this behavior memory is divided into memory generations
  - hold objects of different age
- GC manages every generation separately
  - clears when generation gets too big
  - assuming, that most objects die young
    - GC mostly works on minor collection
      - most objects starts and ends here
    - GC clears what is significant
      - old objects are not here (live for the whole time)

Minor collections

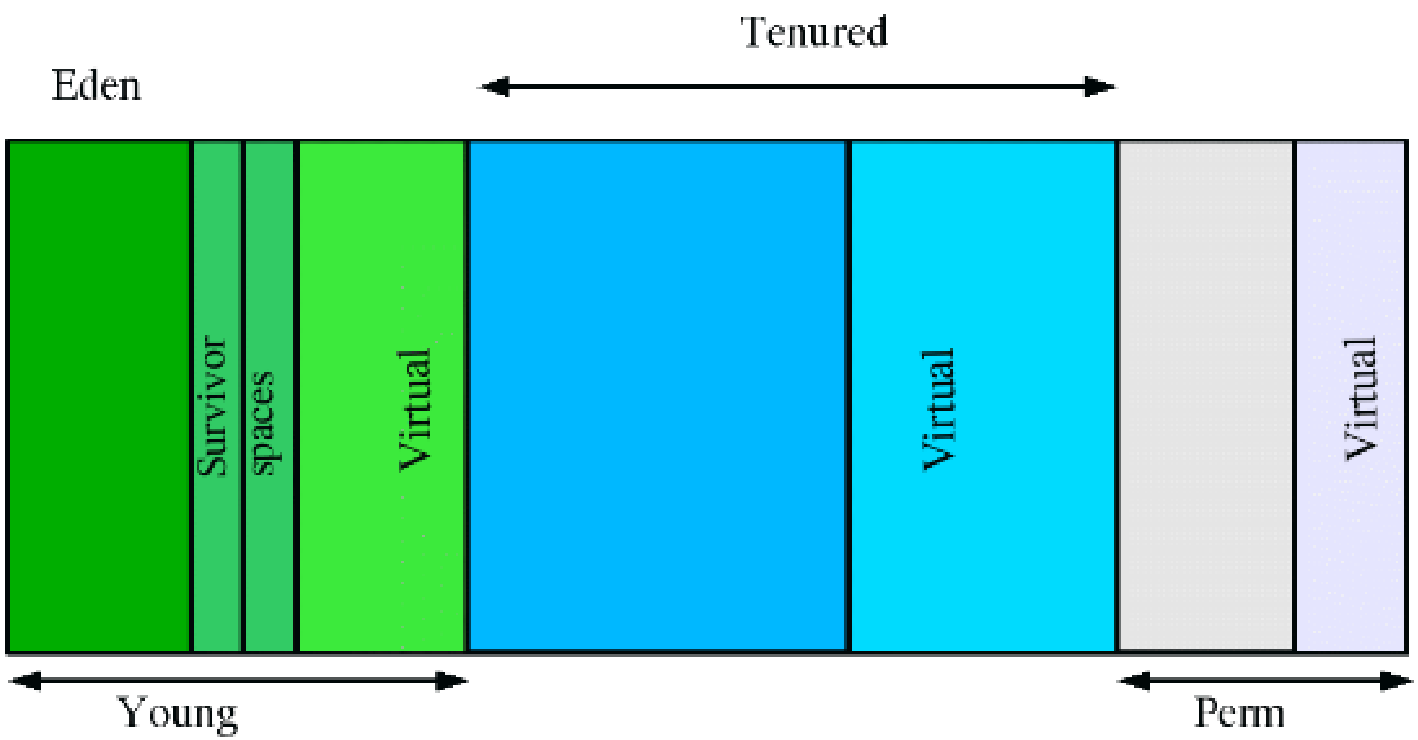
- concerns only young generation
- happens most often
- most important when configuring GC
- young generation full of old objects is cleared fast
- some survived objects get reallocated to older generation (tenured)

Major collections

- covers whole heap
- occur, when whole old generation filled
- takes more time
  - more elements to collect
- System.gc() – request for major collection
  - AVOID

Generations

- standard generation model for all types of GC, besides parallel collector



Generations – behaviour

- during initialization the maximum amount of memory gets virtually reserved
  - but not allocated in physical memory until needed
- address space is divided in yound and old generation
- *permanent* generation used by JVM
  - objects describing methods and classes

Young generation

- young generation consists of
  - Eden
    - youngest objects
    - most of the objects start here
  - two survivor spaces
    - for objects surviving garbage collection
    - one is for objects surviving clearing Eden
    - second one is empty
      - used alternately
      - object surviving next clearing goes to second space
      - objects are copied until aged enough to be moved to old generation

Thoughts on efficiency

- when GC becomes bottleneck configuration is needed
  - full heap size
  - generation sizes
- option -verbose:gc
  - returns data from GC for every clear
  - allows for better tuning
  - example: 2 minor and 1 major collection
  - format: before->after(heap size), GC time

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

  - -XX:+PrintGCDetail shows additional information
  - -XX:+PrintGCTimeStamps adds timestamps

Thoughts on efficiency

- usually have to choose between different measures
  - big young generation space
    - higher throughput
    - also pause time, footprint and promptness
  - small young generation space
    - lowers pauses
    - lowers throughput
- changing the size of one generation does not affect others
- there are no recipes for dimensioning
  - best choice depends on user requirements and how the application uses memory
  - default choice is not always right
    - can be changed through command line options