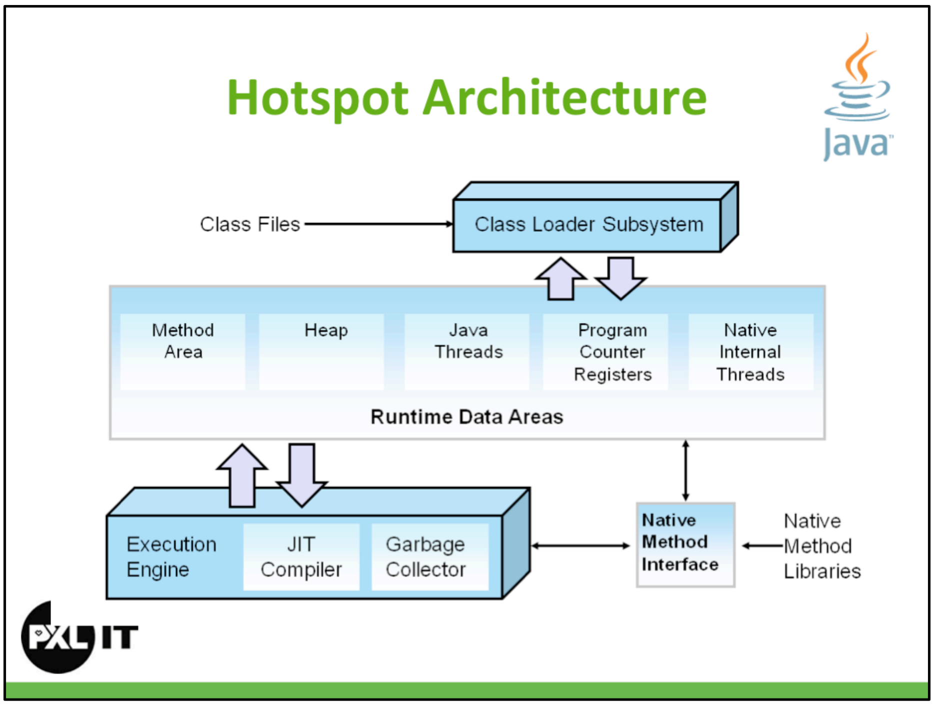
PXL IT

W5: Garbage Collection GC

.NET / Java

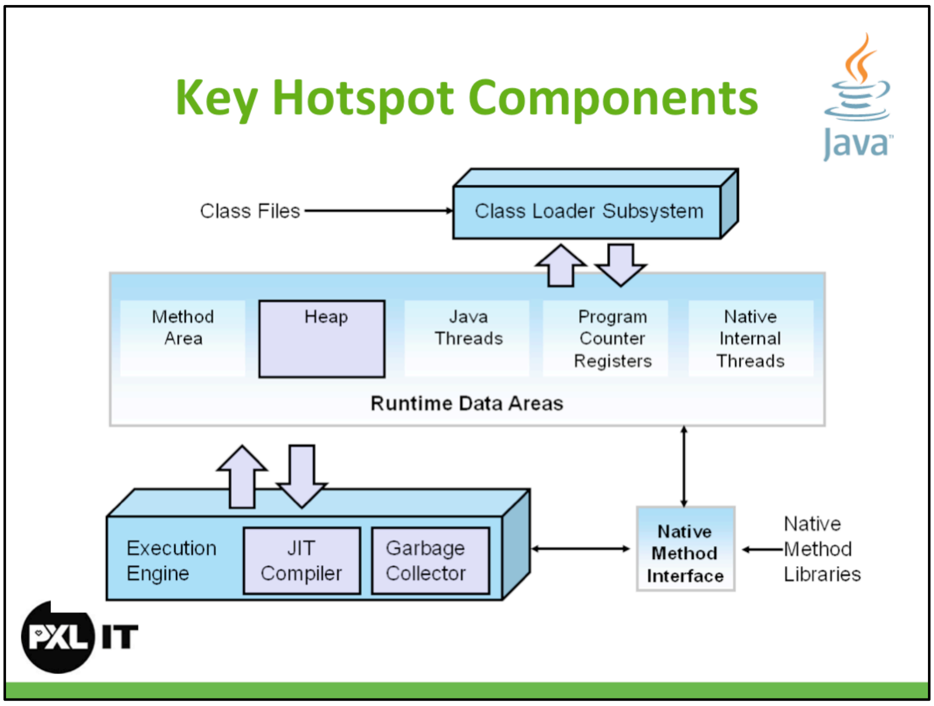
DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Dep. PXL-IT – Elfde-Liniestraat 26 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

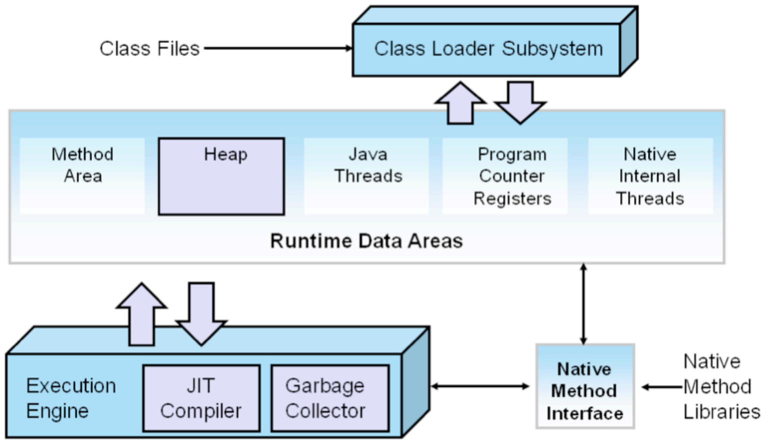

Hotspot Architecture

The HotSpot JVM possesses an architecture that supports a strong foundation of features and capabilities and supports the ability to realize high performance and massive scalability. For example, the HotSpot JVM JIT compilers generate dynamic optimizations. In other words, they make optimization decisions while the Java application is running and generate high-performing native machine instructions targeted for the underlying system architecture. In addition, through the maturing evolution and continuous engineering of its runtime environment and multithreaded garbage collector, the HotSpot JVM yields high scalability on even the largest available computer systems.

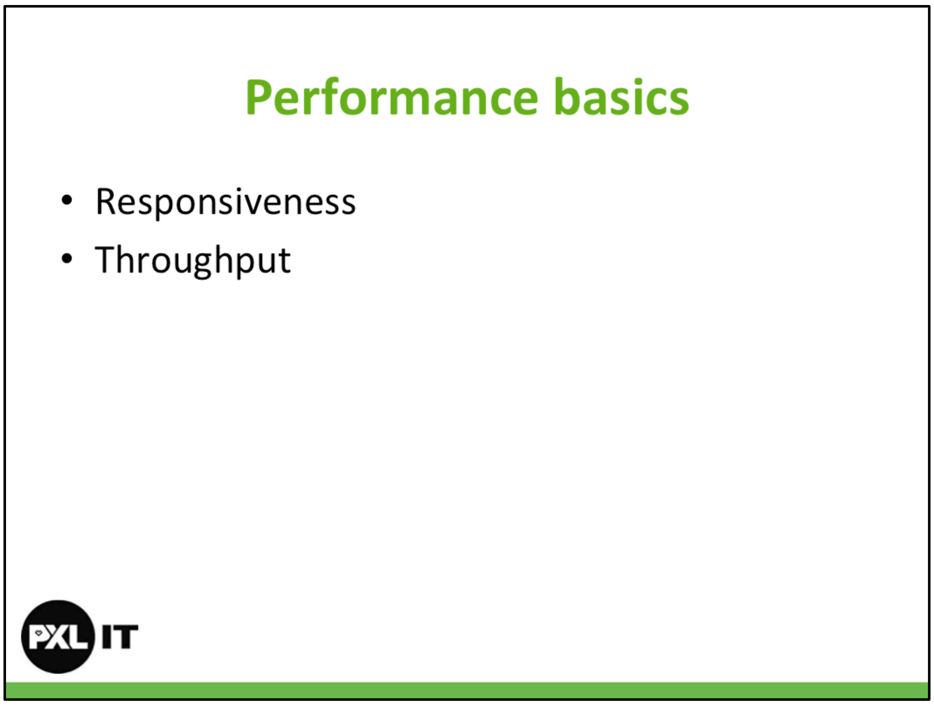
The main components of the JVM include the classloader, the runtime data areas, and the execution engine.



Key Hotspot Components



There are three components of the JVM that are focused on when tuning performance. The *heap* is where your object data is stored. This area is then managed by the garbage collector selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The JIT compiler also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.



Performance basics

- Responsiveness
- Throughput

Responsiveness

Responsiveness refers to how quickly an application or system responds with a requested piece of data. Examples include:

How quickly a desktop UI responds to an event

How fast a website returns a page

How fast a database query is returned

For applications that focus on respsonsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

Throughput

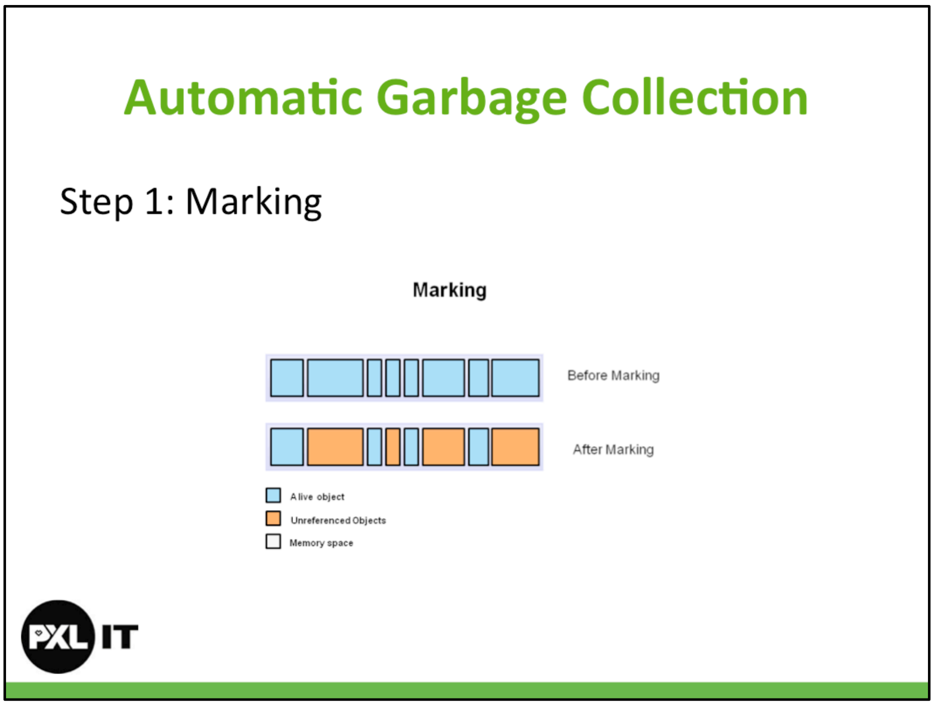
Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:

The number of transactions completed in a given time.

The number of jobs that a batch program can complete in an hour.

The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick



Automatic Garbage Collection

Step 1: Marking

What is Automatic Garbage Collection?

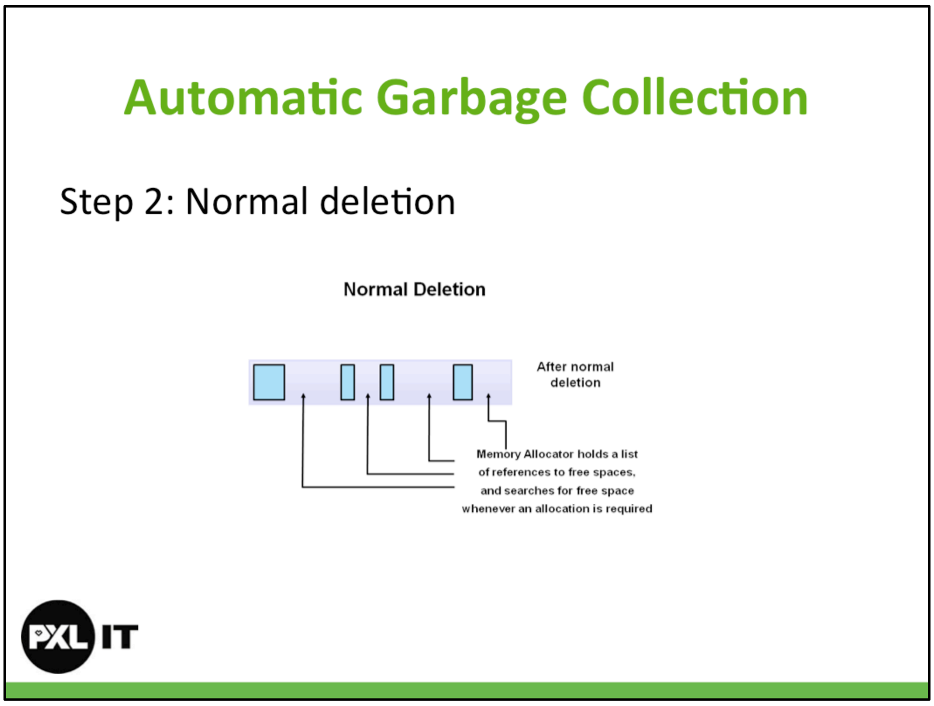
Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

In a programming language like C, allocating and deallocating memory is a manual process. In Java, process of deallocating memory is handled automatically by the garbage collector. The basic process can be described as follows.

Step 1: Marking

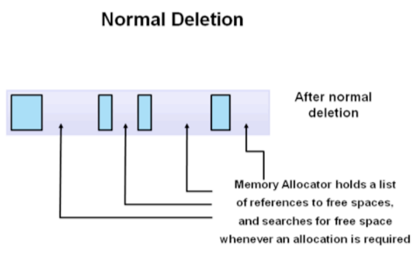
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.

Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.



Automatic Garbage Collection

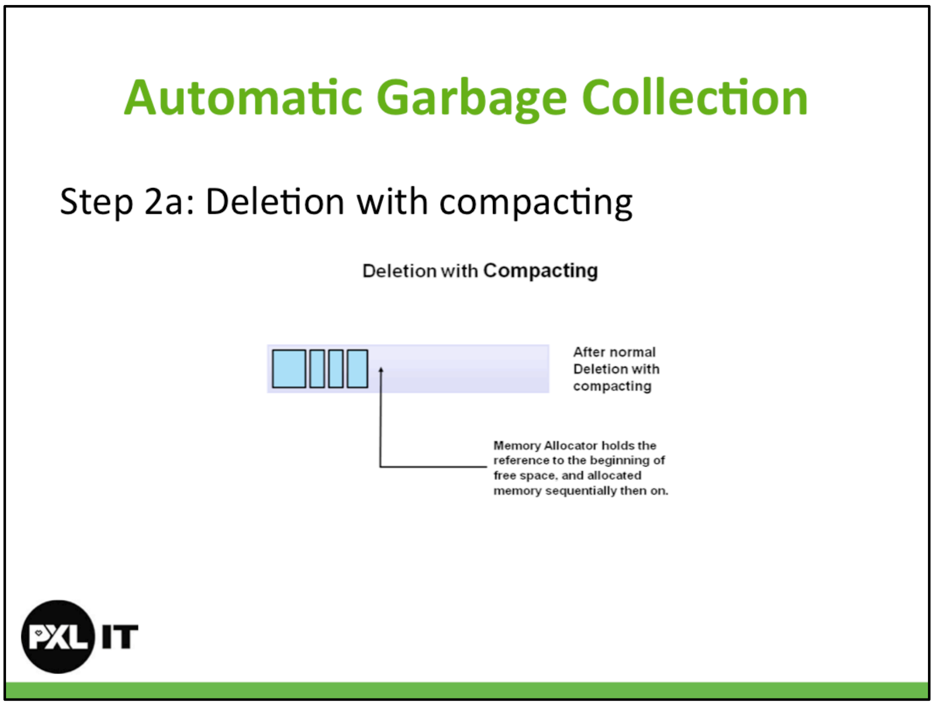
Step 2: Normal deletion


Normal Deletion
After normal deletion
Memory Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required

Step 2: Normal Deletion

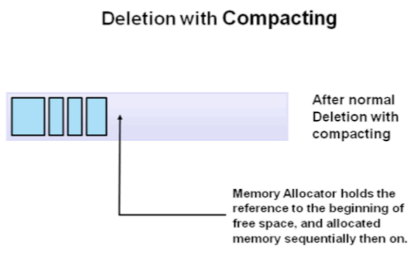
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space.

The memory allocator holds references to blocks of free space where new object can be allocated.



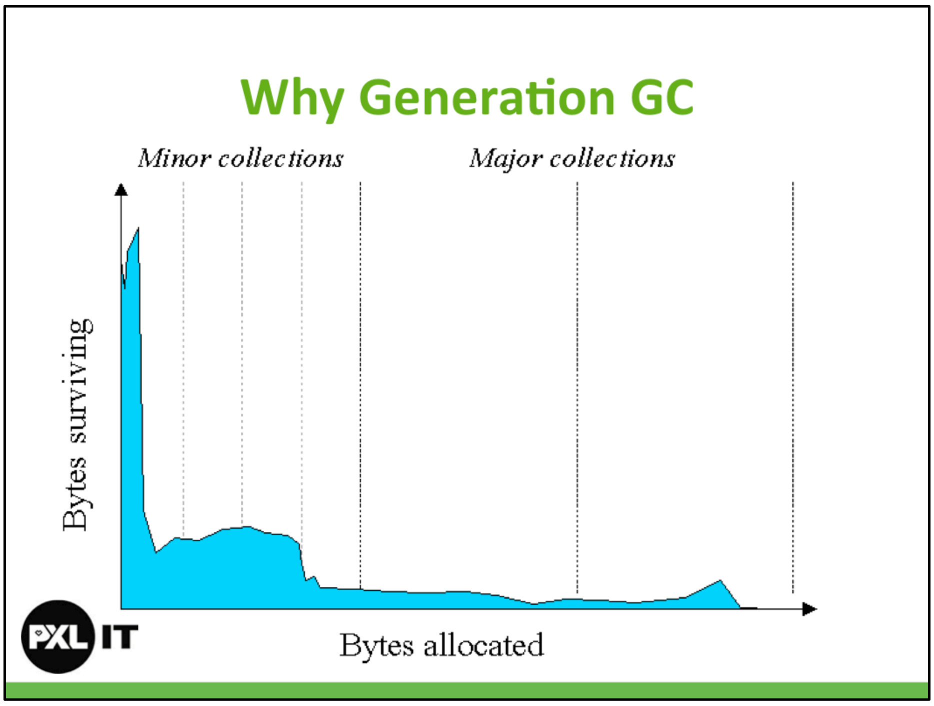
Automatic Garbage Collection

Step 2a: Deletion with compacting


Deletion with Compacting
After normal Deletion with compacting
Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.

Step 2a: Deletion with Compacting

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

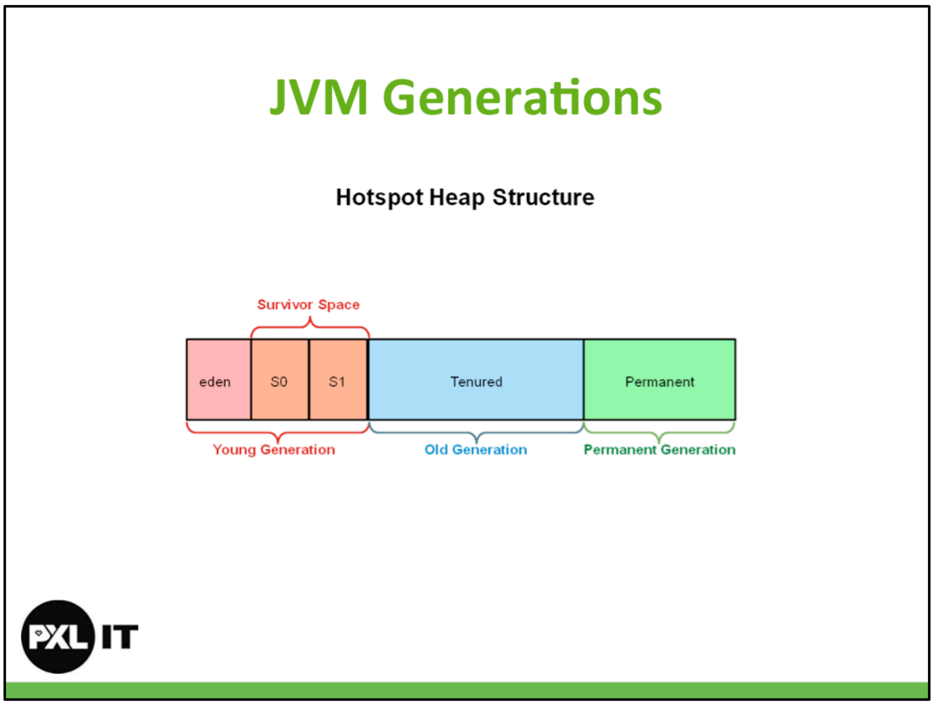


Why Generation GC

As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

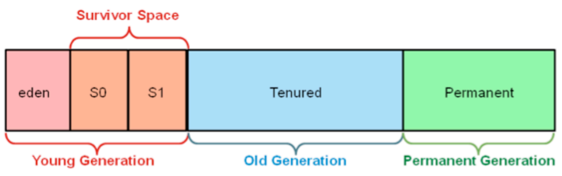
Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.

As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.



JVM Generations

Hotspot Heap Structure



The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a ***minor garbage collection***. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

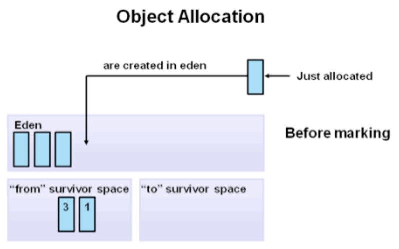
Stop the World Event - All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are *always* Stop the World events.

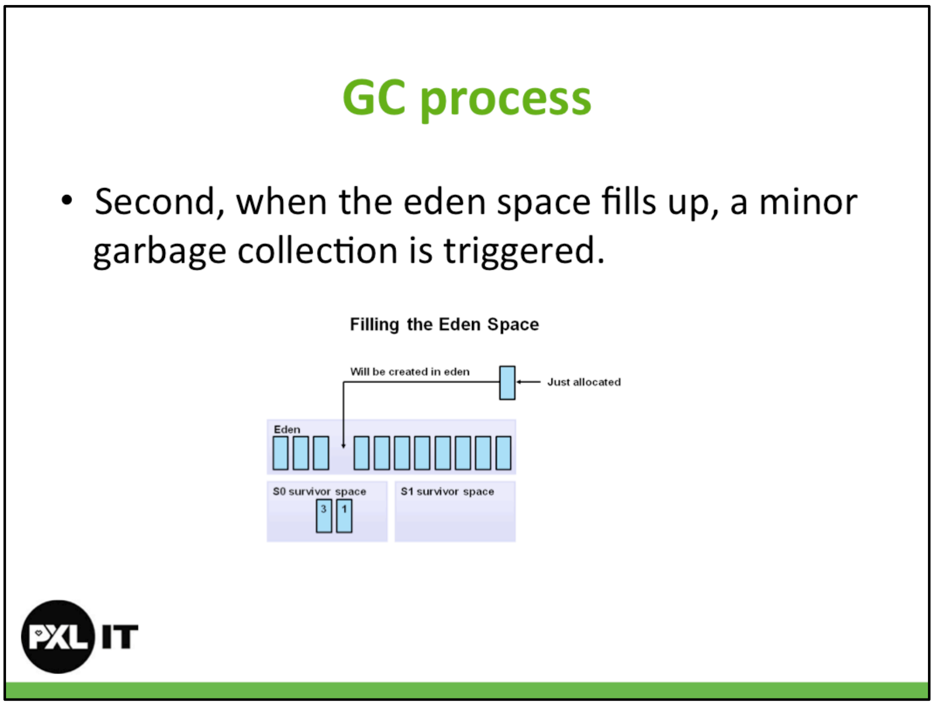
The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a ***major garbage collection***.

Major garbage collection are also Stop the World events. Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

GC process

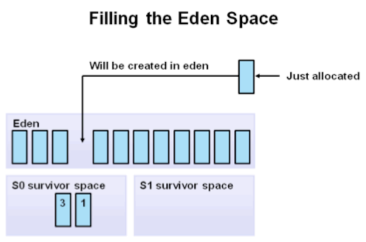
- First, any new objects are allocated to the eden space. Both survivor spaces start out empty.





GC process

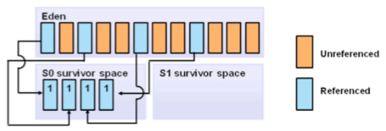
- Second, when the eden space fills up, a minor garbage collection is triggered.

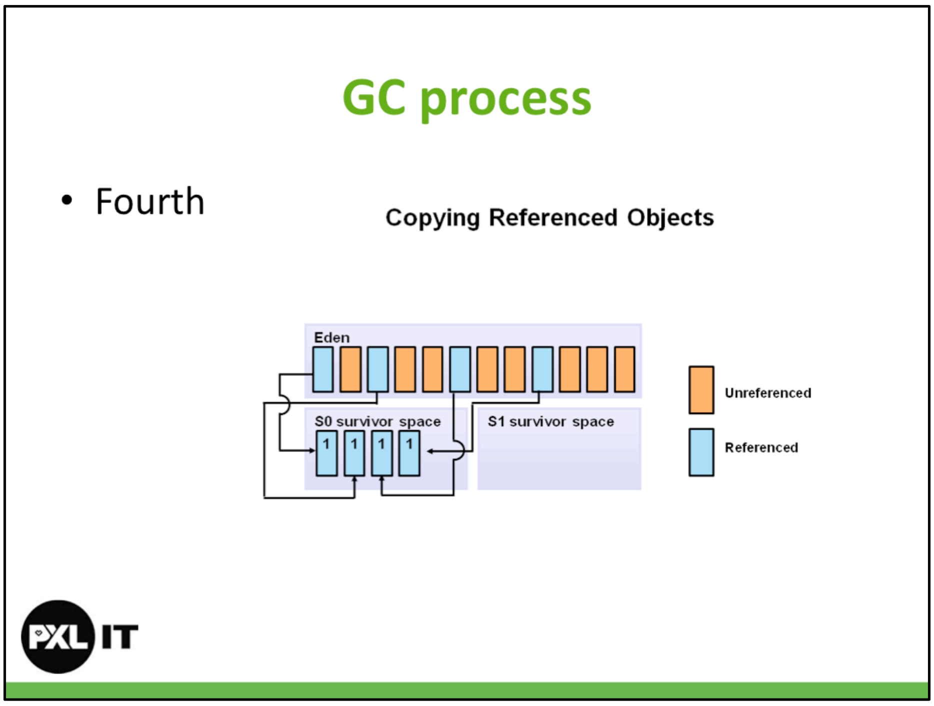


GC process

- Third, referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.

Copying Referenced Objects

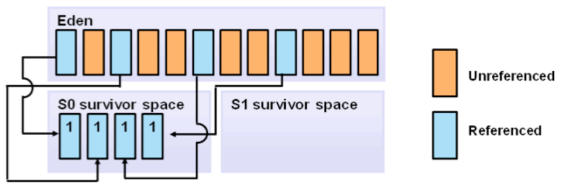




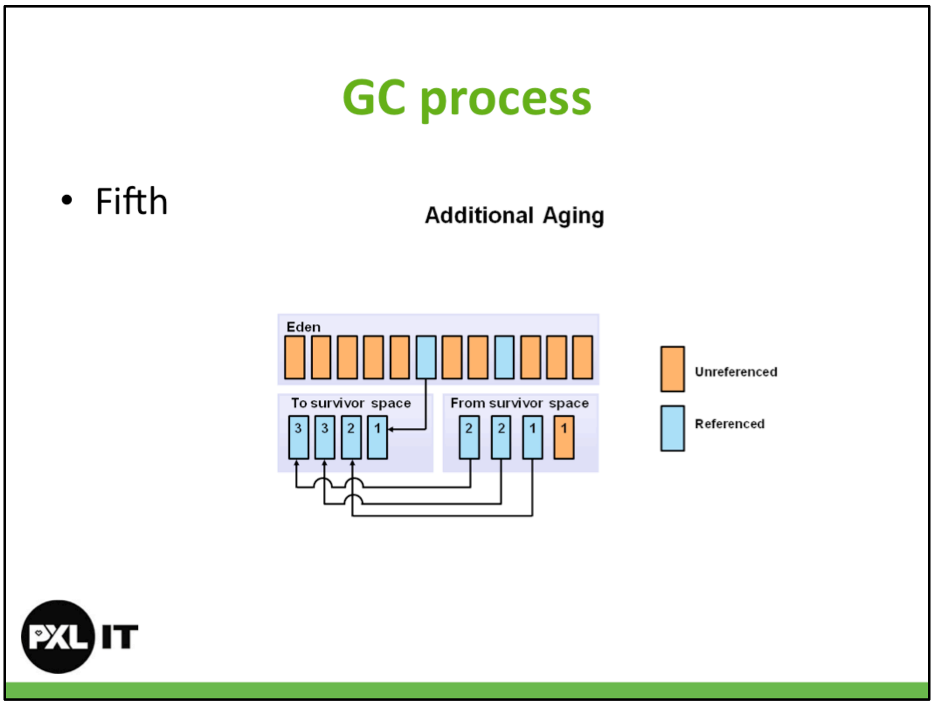
GC process

- Fourth

Copying Referenced Objects

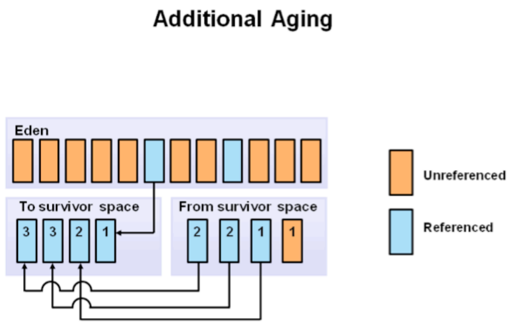


At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.

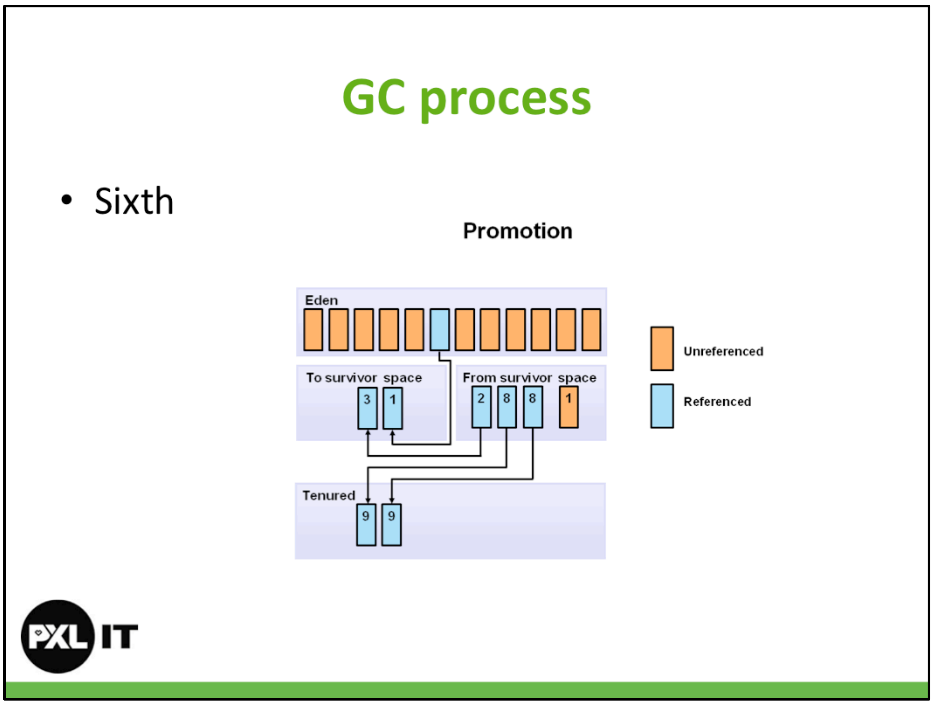


GC process

- Fifth

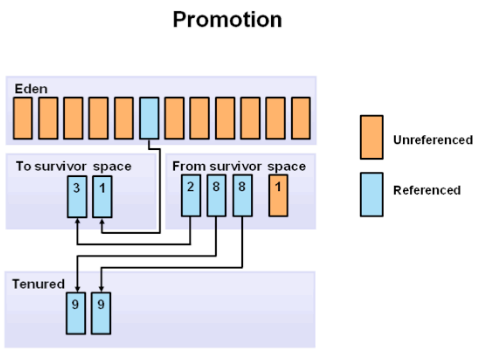


At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.

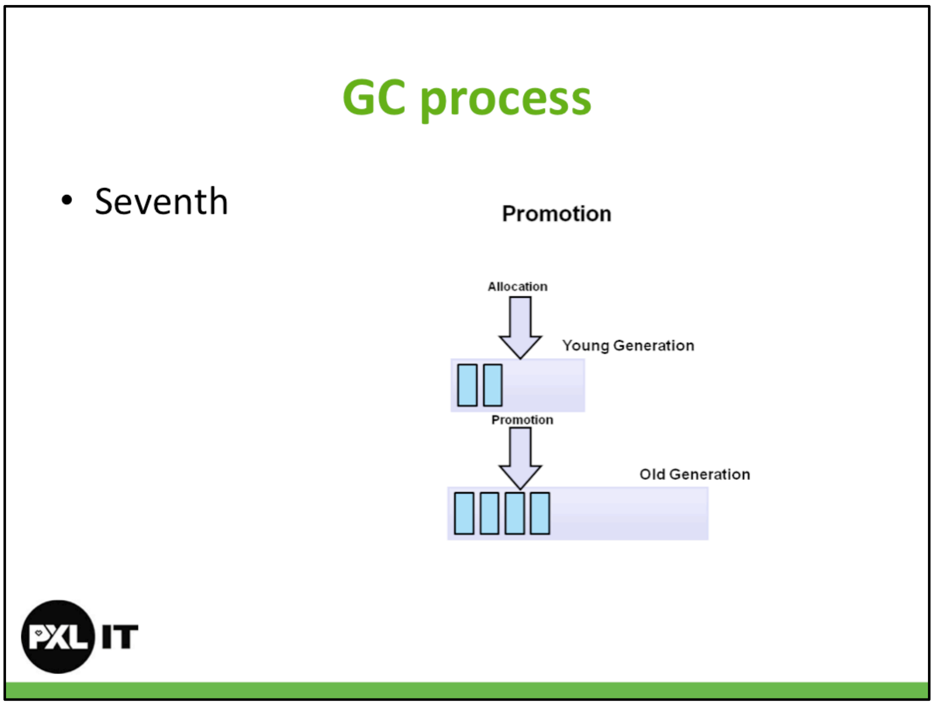


GC process

- Sixth

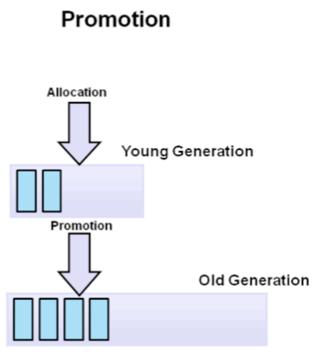


This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.

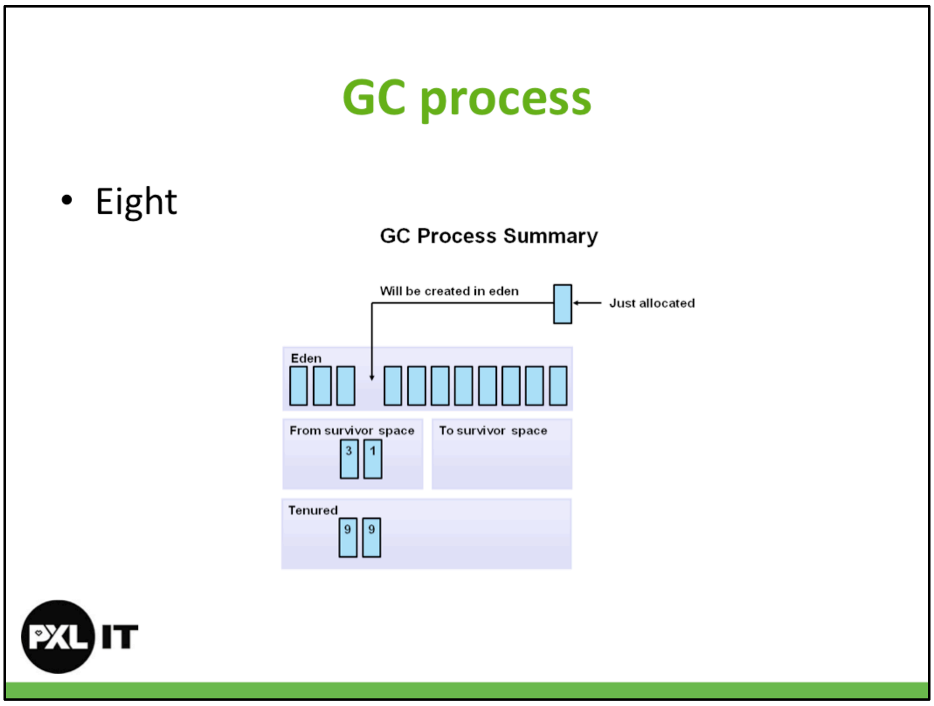


GC process

- Seventh

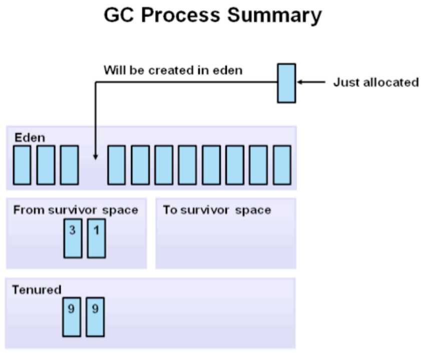


As minor GCs continue to occure objects will continue to be promoted to the old generation space.

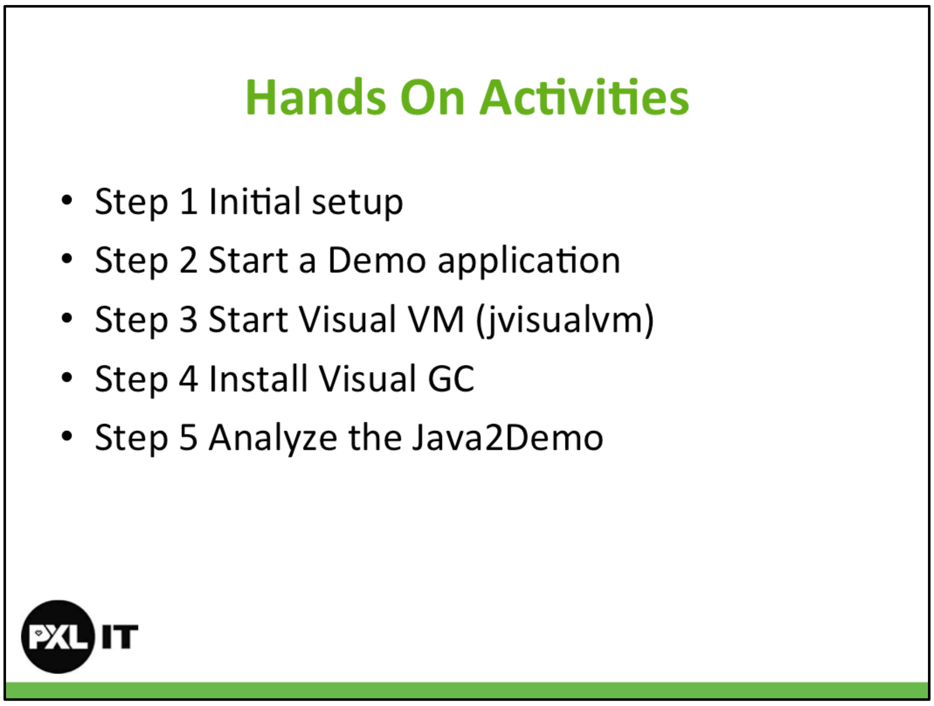


GC process

- Eight



So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.



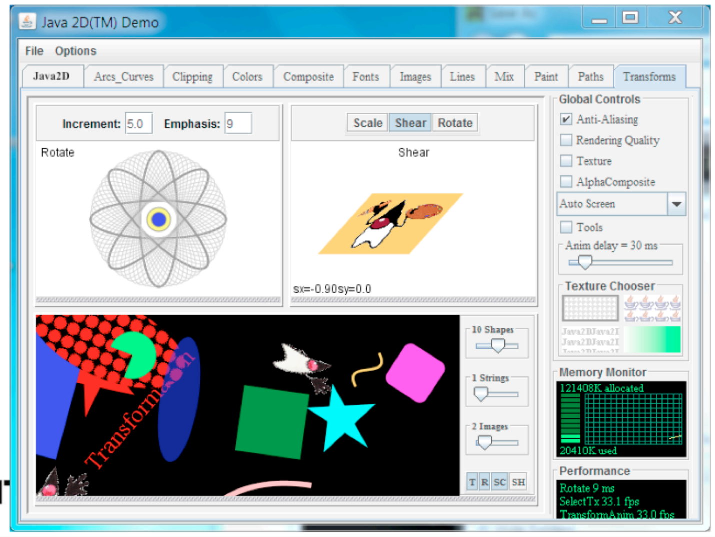
Hands On Activities

- Step 1 Initial setup
- Step 2 Start a Demo application
- Step 3 Start Visual VM (jvisualvm)
- Step 4 Install Visual GC
- Step 5 Analyze the Java2Demo

PXL IT

You have seen the garbage collection process using a series of pictures. Now it is time to experience and explore the process live. In this activity, you will run a Java application and analyze the garbage collection process using Visual VM. The Visual VM program is included with the JDK and allows developers to monitor various aspects of a running JVM.

Hands On Activities



Increment: 5.0 Emphasis: 9

sx=-0.90sy=0.0

10 Shapes

1 Strings

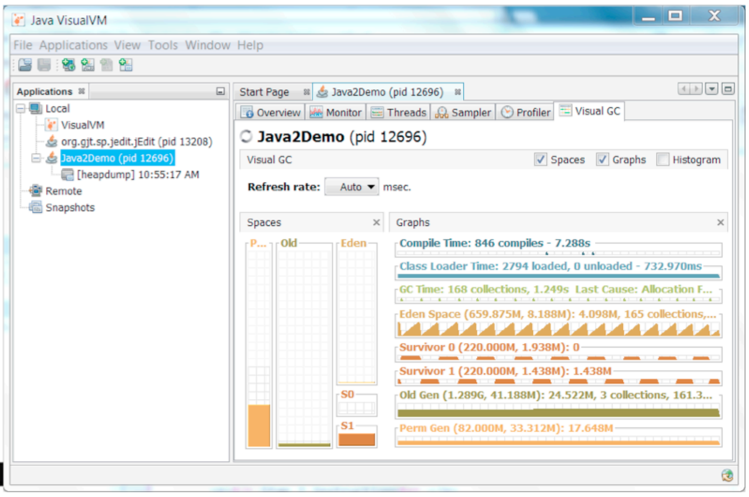
2 Images

[✓] Anti-Aliasing
☐ Rendering Quality
☐ Texture
☐ AlphaComposite
Auto Screen
☐ Tools
Anim delay = 30 ms

121408K allocated
20410K used

Rotate 9 ms
SelectTx 33.1 fps
TransformAnim 33.0 fps

Hands On Activities



Java2Demo (pid 12696)

Visual GC [✓ Spaces] [✓ Graphs] Histogram

Refresh rate: Auto msec.

Compile Time: 846 compiles - 7.288s
Class Loader Time: 2794 loaded, 0 unloaded - 732.970ms
GC Time: 168 collections, 1.249s Last Cause: Allocation F...
Eden Space (659.875M, 8.188M): 4.098M, 165 collections,...
Survivor 0 (220.000M, 1.938M): 0
Survivor 1 (220.000M, 1.438M): 1.438M
Old Gen (1.289G, 41.188M): 24.522M, 3 collections, 161.3...
Perm Gen (82.000M, 33.312M): 17.648M