



# JVM Memory Model and GC

## Developer Community Support

Fairoz Matte  
Principle Member Of Technical Staff  
Java Platform Sustaining Engineering,



# Agenda

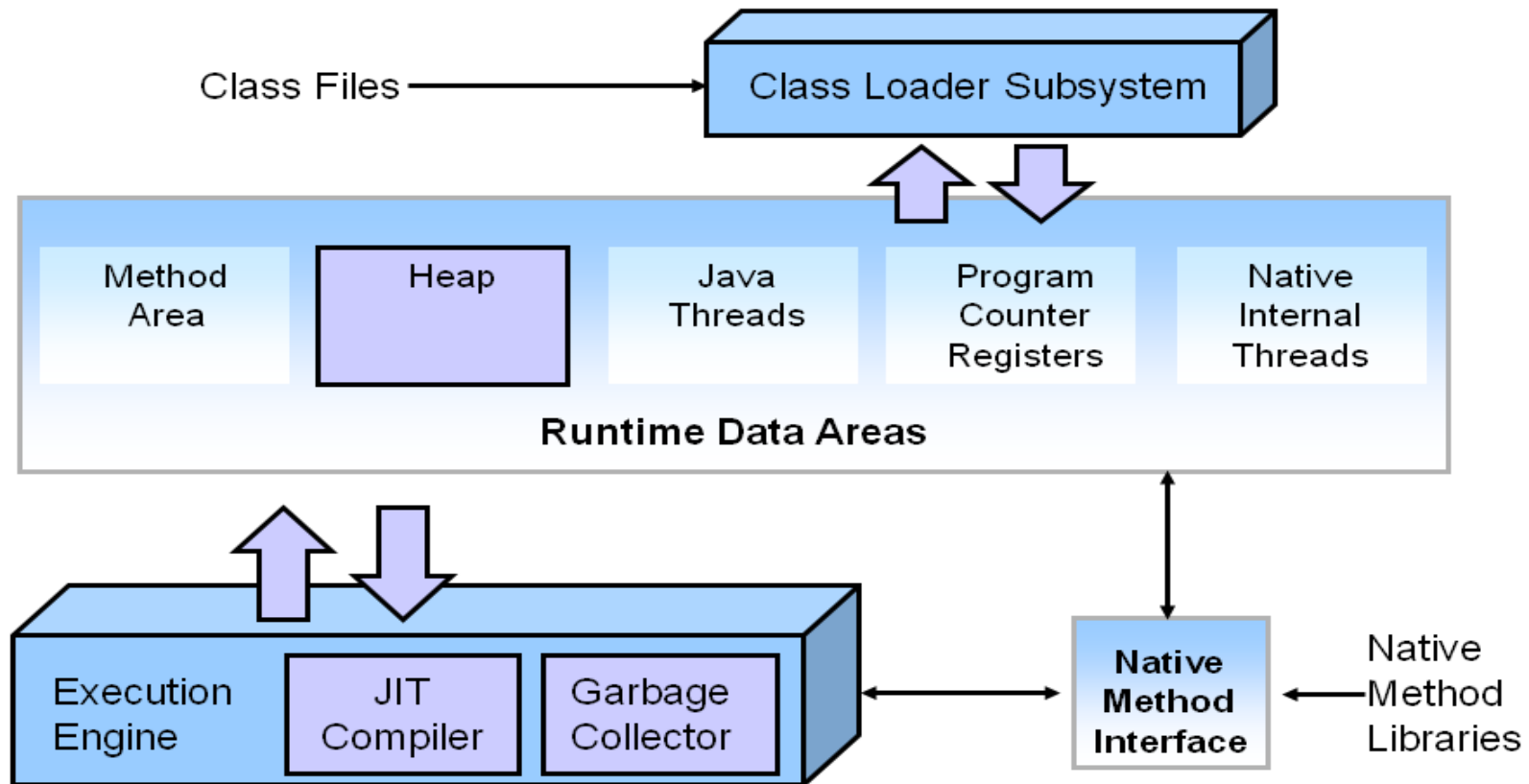
- 1 ➤ What is JVM?
- 2 ➤ JVM Architecture
- 3 ➤ Hotspot Memory Structure
- 4 ➤ Garbage Collection Process
- 5 ➤ Demo Application
- 6 ➤ Garbage collection Types

# What is JVM?

- The Java Virtual Machine (JVM) is an abstract computing machine.
- This way, Java programs are written to the same set of interfaces and libraries.
- Each JVM implementation for a specific operating system, translates the Java programming instructions into instructions and commands that run on the local operating system. Java programs achieve platform independence.
- The Java virtual machine knows nothing of the Java programming language, only of a particular binary format, the class file format.

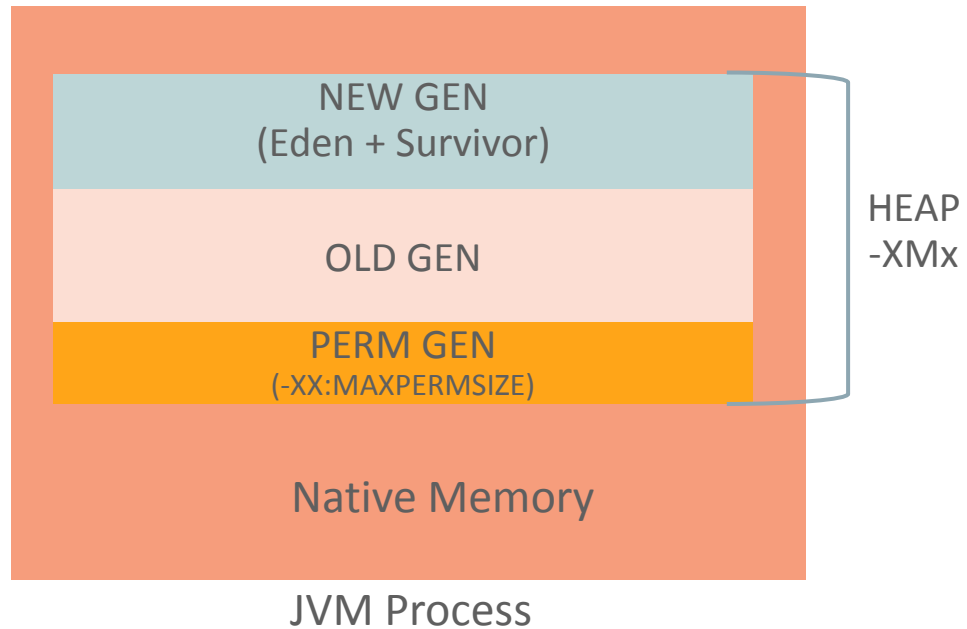
# JVM Architecture

## Key HotSpot JVM Components

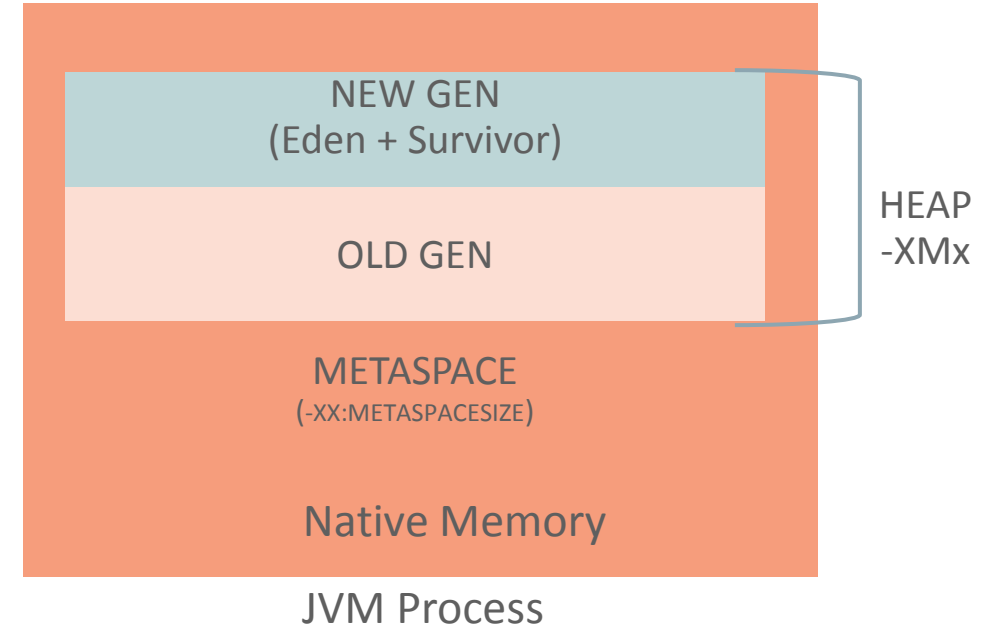


# Hotspot Memory Structure

Pre Java 8



Java 8



# Hotspot Memory Structure - Terminologies

- 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.

# Hotspot Memory Structure - Terminologies

- 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
- **Metaspace** : A native memory to store class meta-data information and that grows automatically & garbage collected.
- **Native Memory**: Native memory is the memory which is available to a process, e.g. the Java process. Native memory is controlled by the operating system and based on physical memory



# Garbage Collection Process

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.


## Step 1: Marking



Before Marking

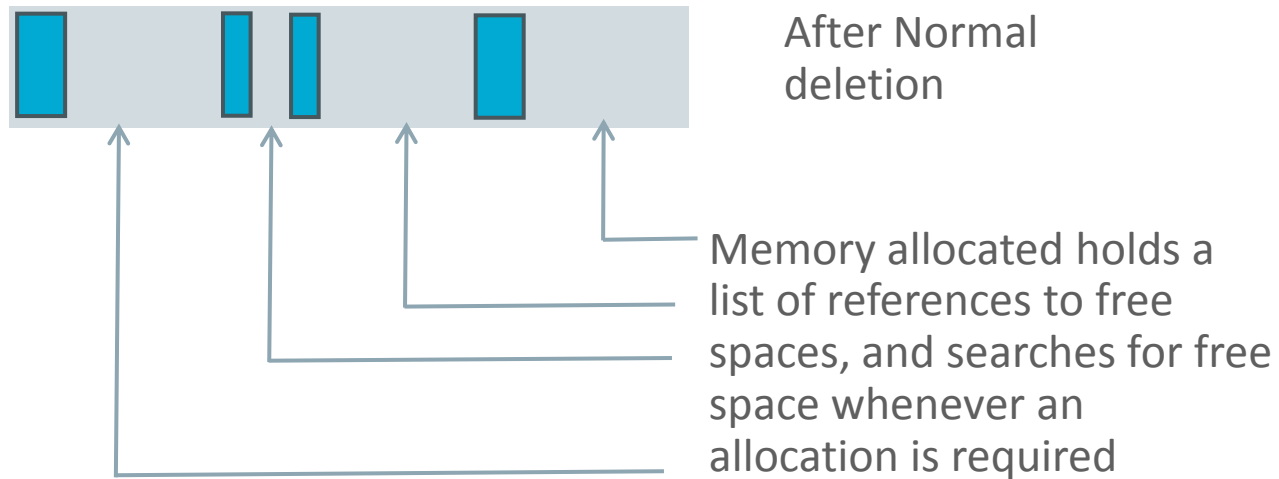


After Marking

-  Live Object
-  Unreferenced Object

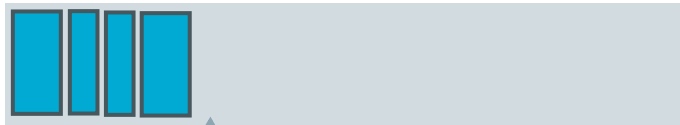
# Garbage Collection Process

## Step 2: Normal Deletion



# Garbage Collection Process

## Step 2a: Deletion with Compaction

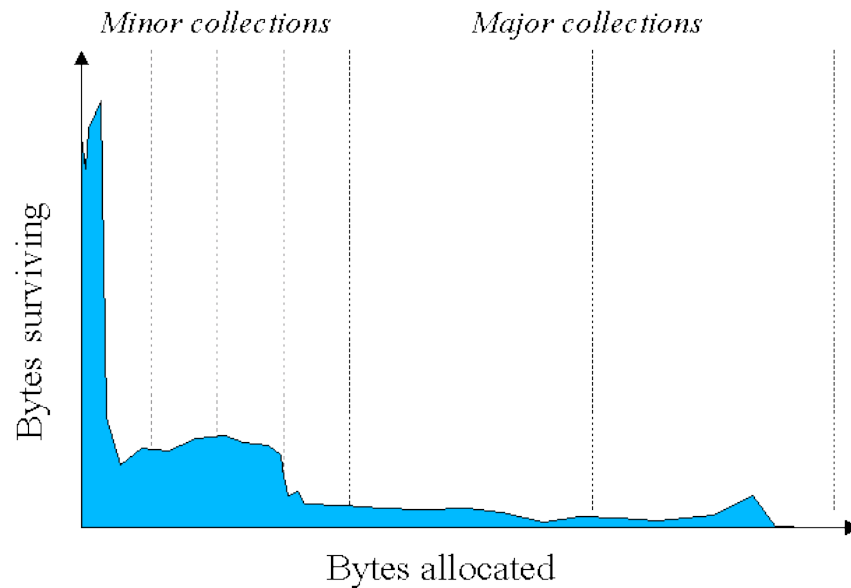


After Normal deletion with  
Compaction

Memory allocated holds the reference  
to the beginning of free space, and  
allocated memory sequentially

# Generational Garbage Collection Process

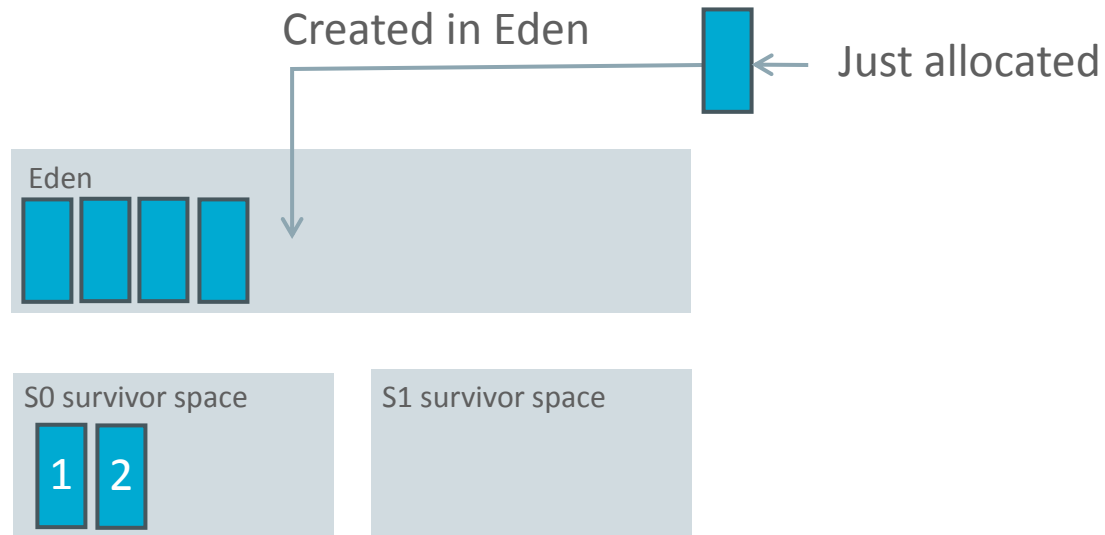
- Heap is separated into different generations
- More and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time
- However, empirical analysis of applications has shown that most objects are short lived



# Generational Garbage Collection Process

## Object Allocation

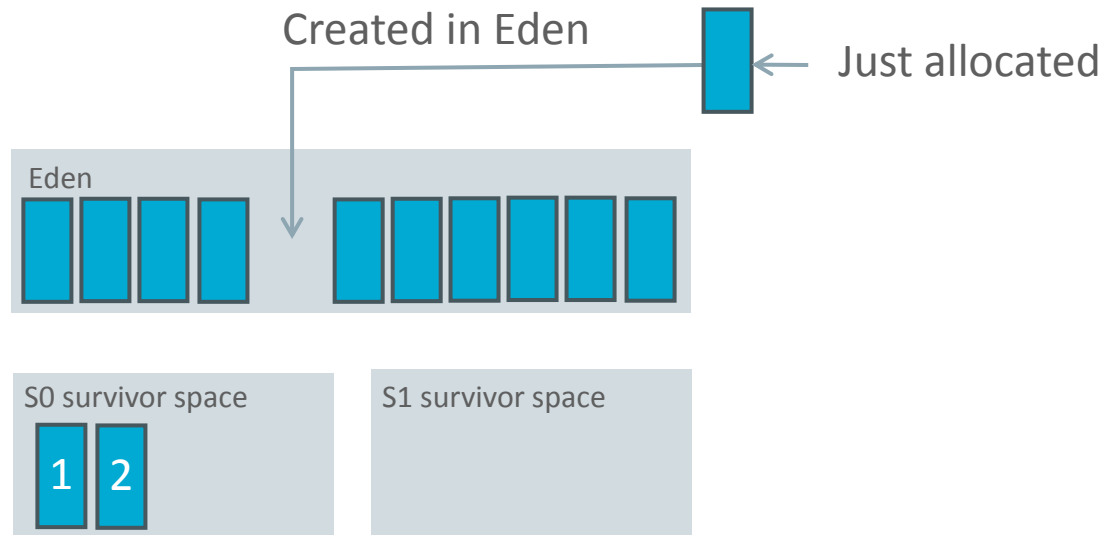
First, any new objects are allocated to the eden space. Both survivor spaces start out empty.



# Generational Garbage Collection Process

## Filling the eden space

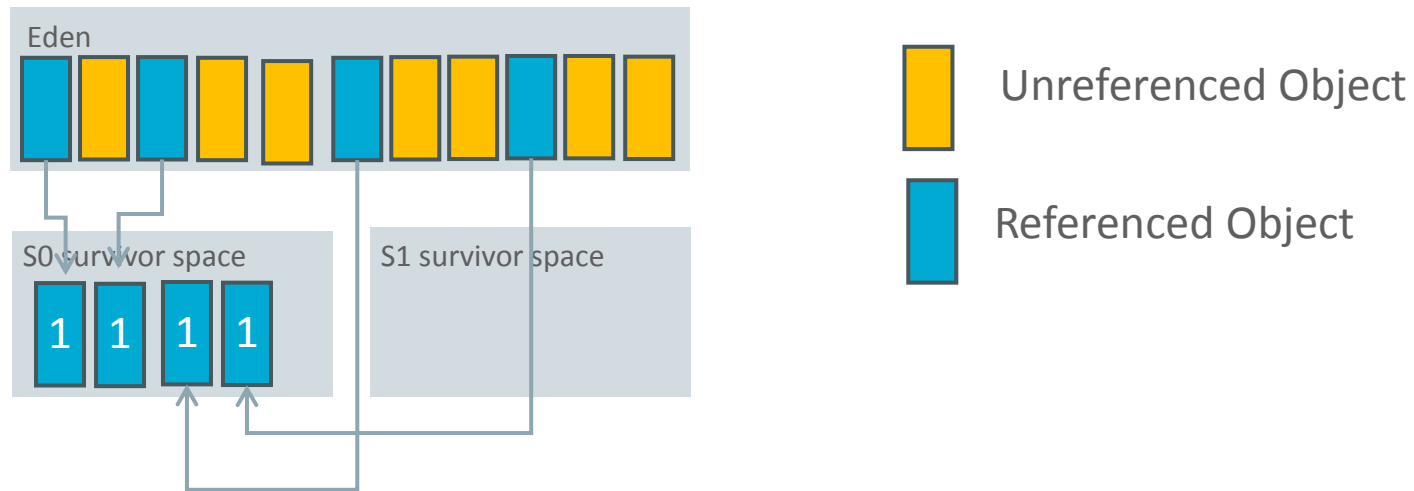
When the eden space fills up, a minor garbage collection is triggered.



# Generational Garbage Collection Process

## Copying referenced objects

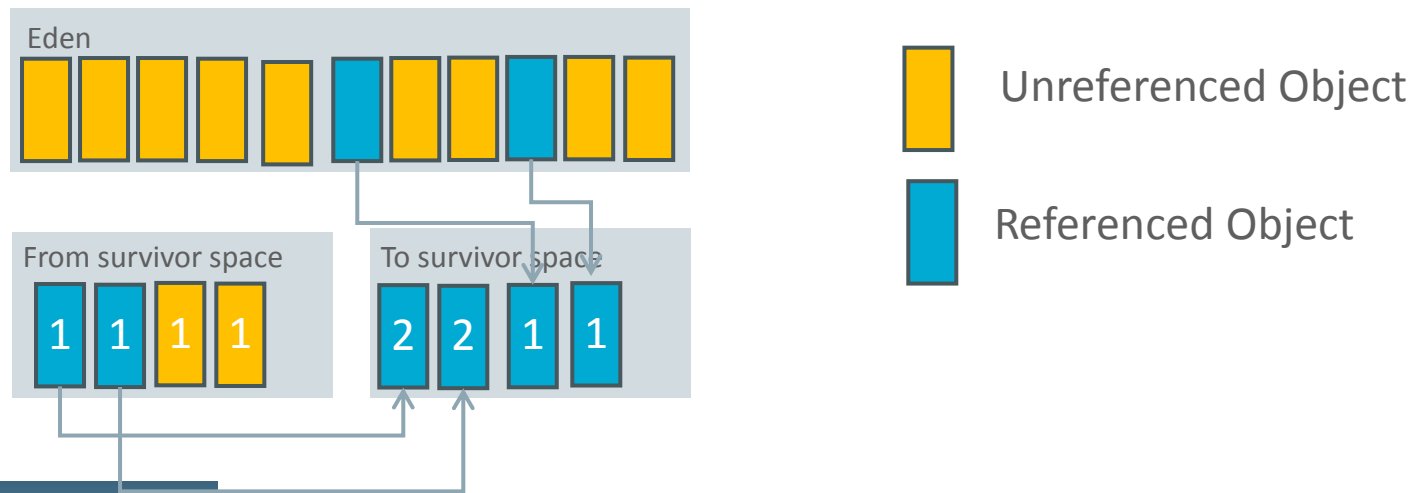
Referenced objects are moved to the first survivor space.  
Unreferenced objects are deleted when the eden space is cleared..



# Generational Garbage Collection Process

## Object Aging

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. ...

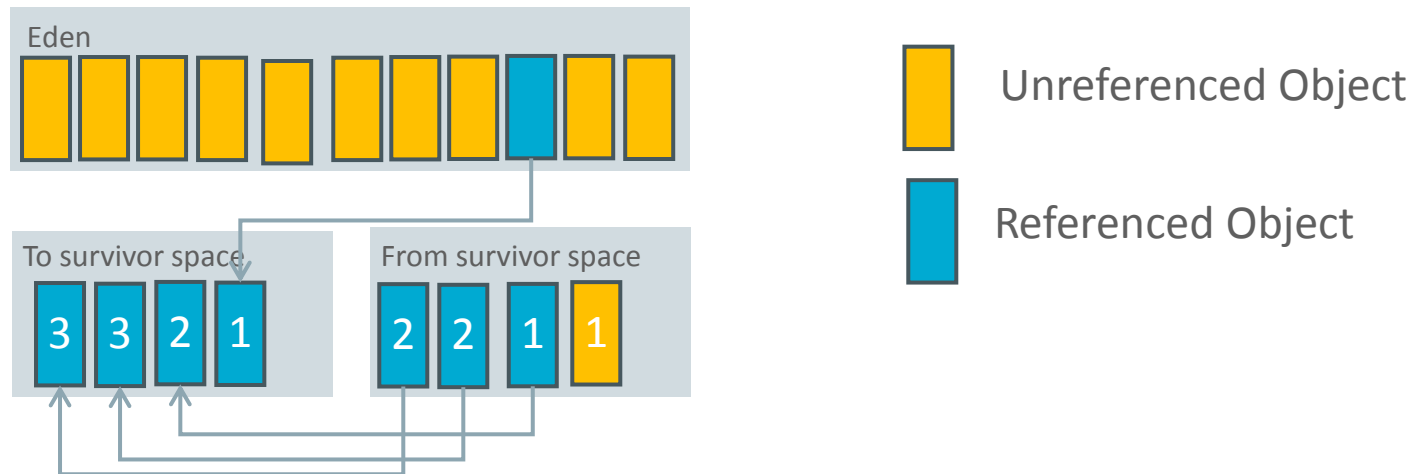




# Generational Garbage Collection Process

## Additional Aging

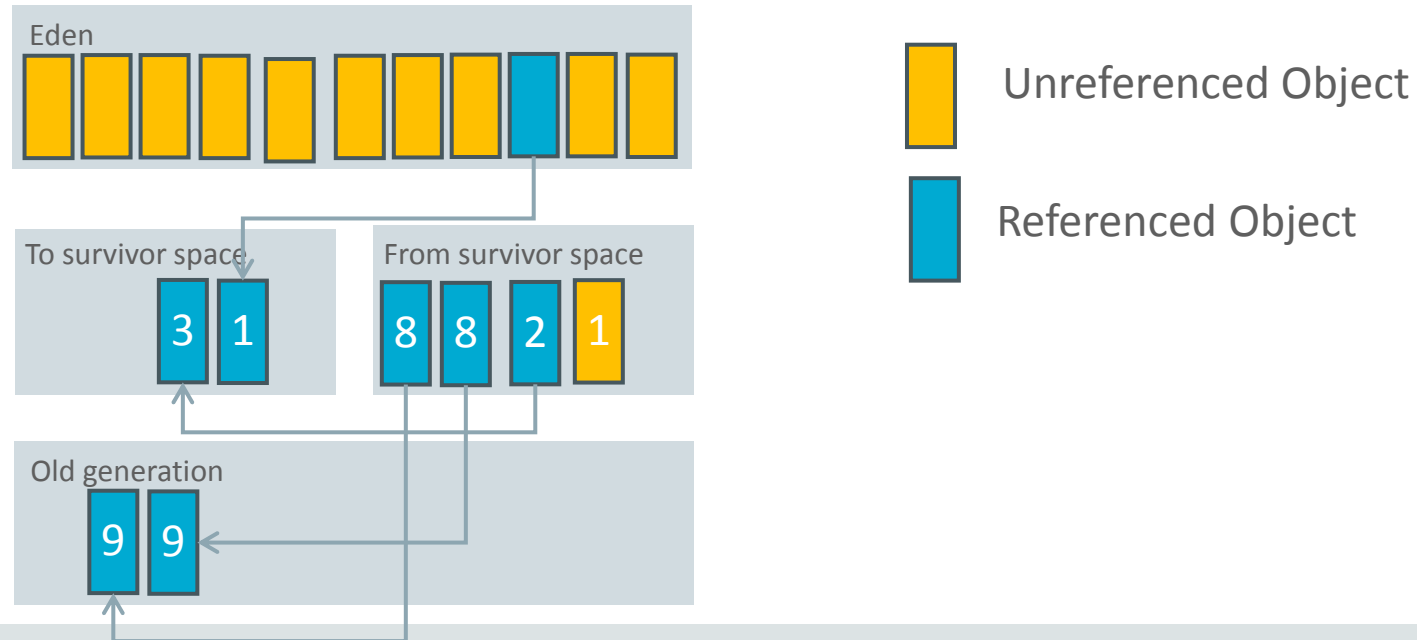
At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



# Generational Garbage Collection Process

## Promotion

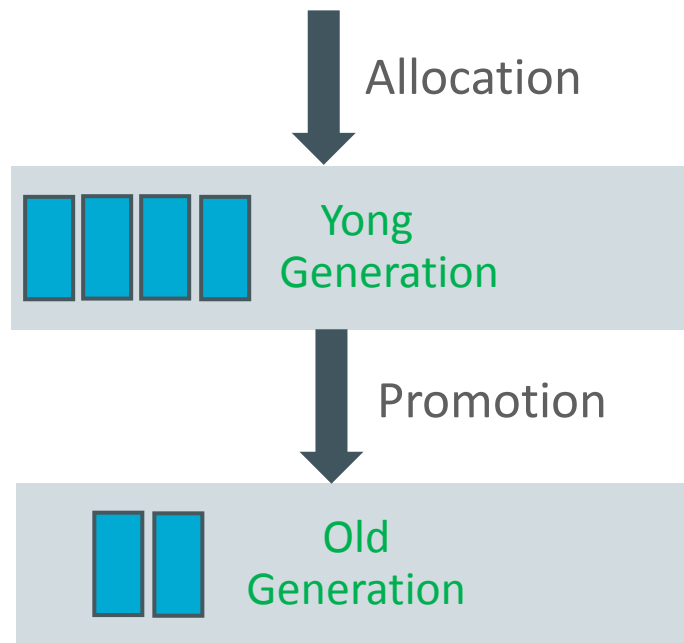
This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.



# Generational Garbage Collection Process

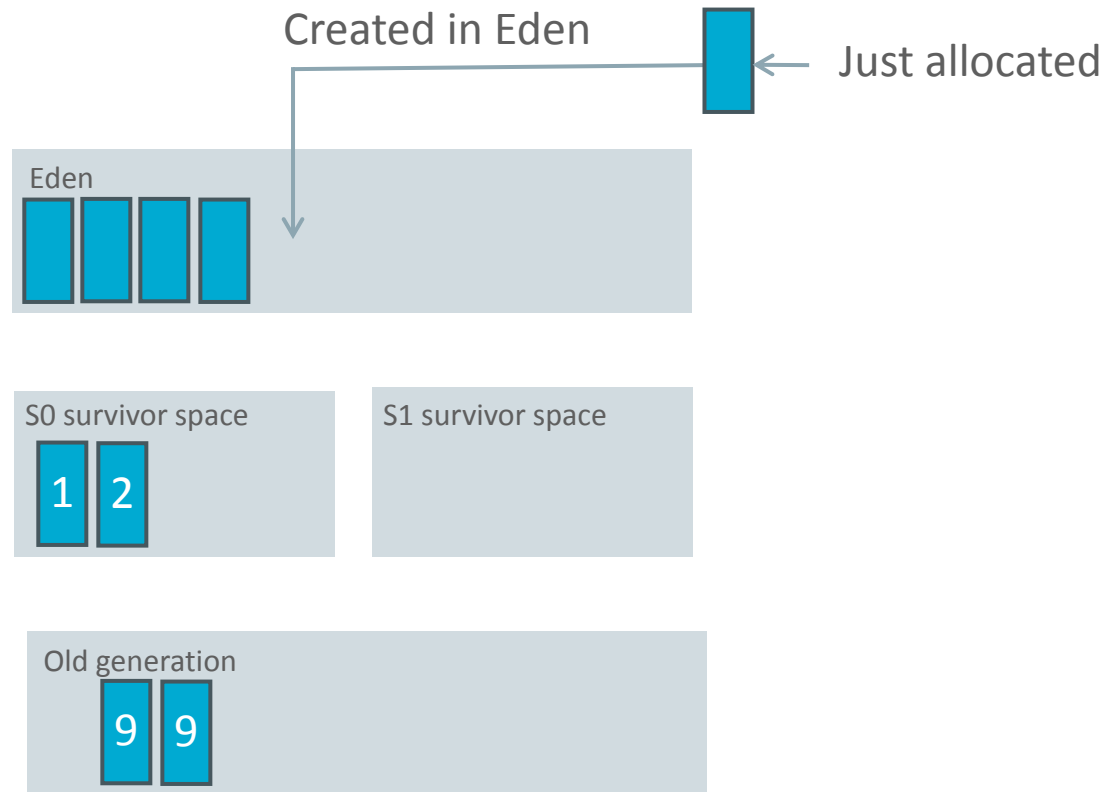
## Promotion

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



# Generational Garbage Collection Process

## GC Process Summary



# Generational Garbage Collection Process

## Demo

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.

# Garbage Collection – Demo Application

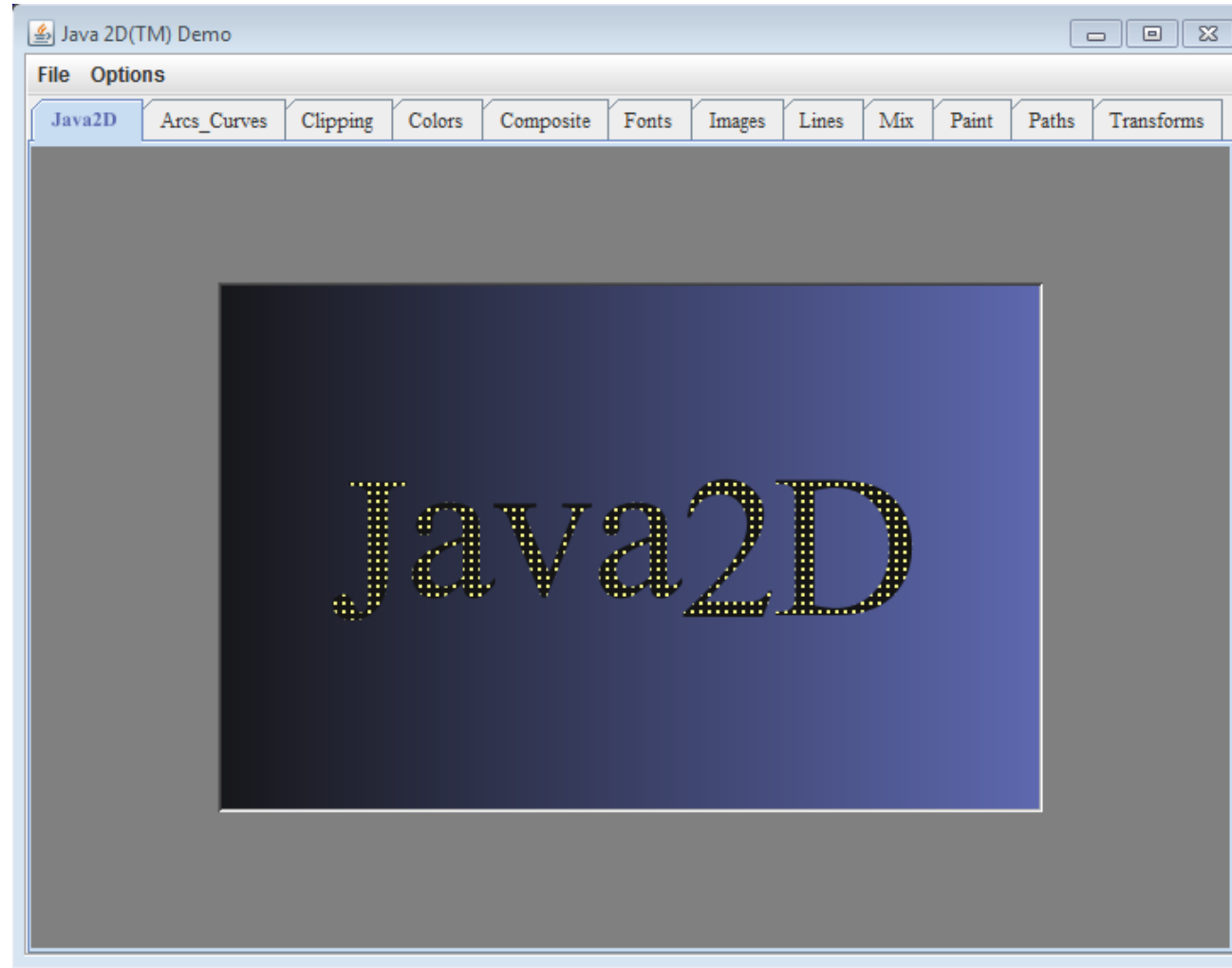
## Start a Demo Application

With the Java JDK and demos installed, you can now run demo application that will be analyzed in this activity. For this example, the demos are installed in D:\2015\jdk-8u101-windows-x64-demos

The demo application is a 2D graphics demo. To execute it type: `java -Xmx12m -Xms3m -Xmn1m XX:+UseSerialGC -jar D:\2015\jdk-8u101-windows-x64-demos\jdk1.8.0_101\demo\jfc\Java2D\Java2demo.jar`

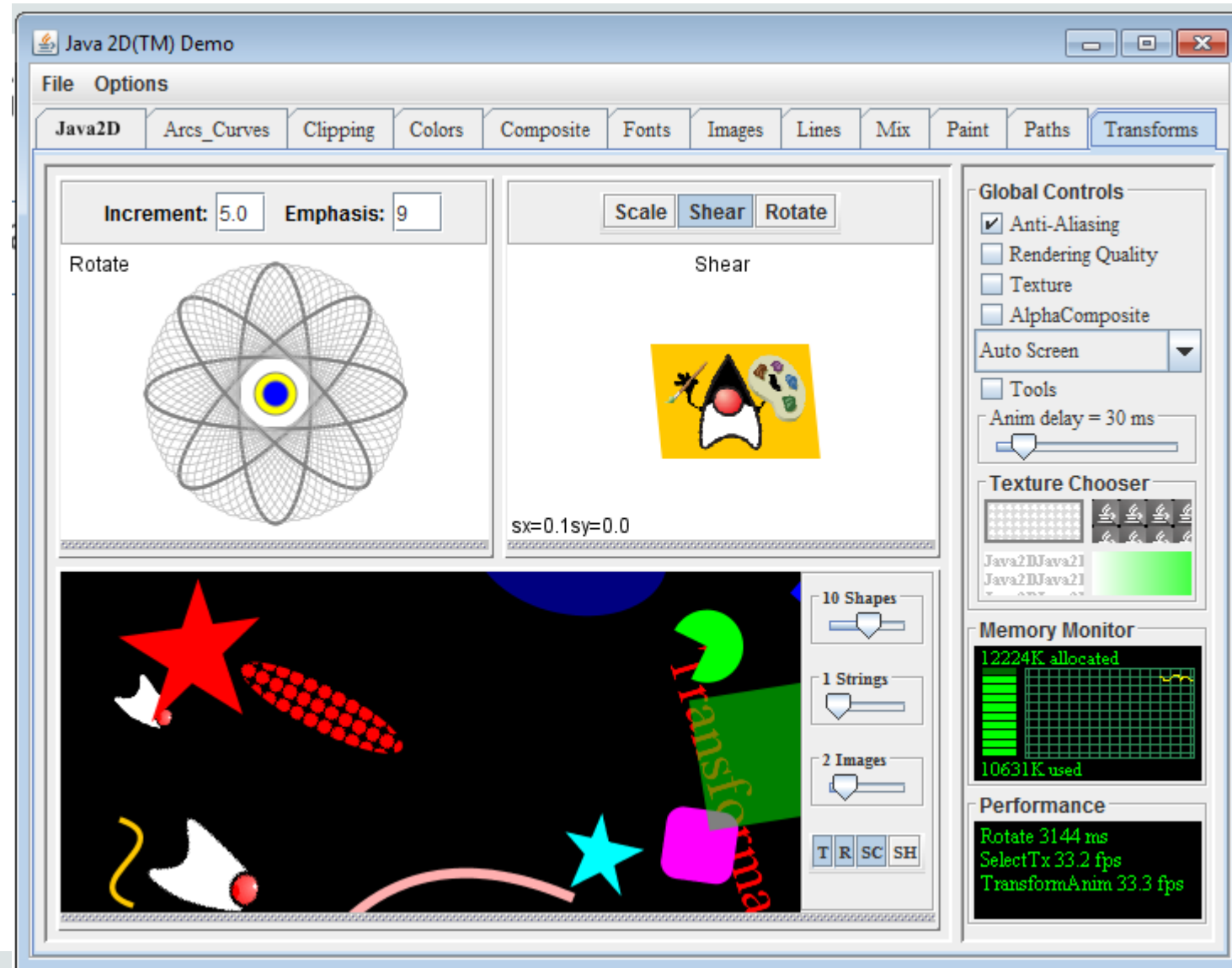
# Garbage Collection – Demo Application

## Java 2D Demo



# Garbage Collection – Demo Application

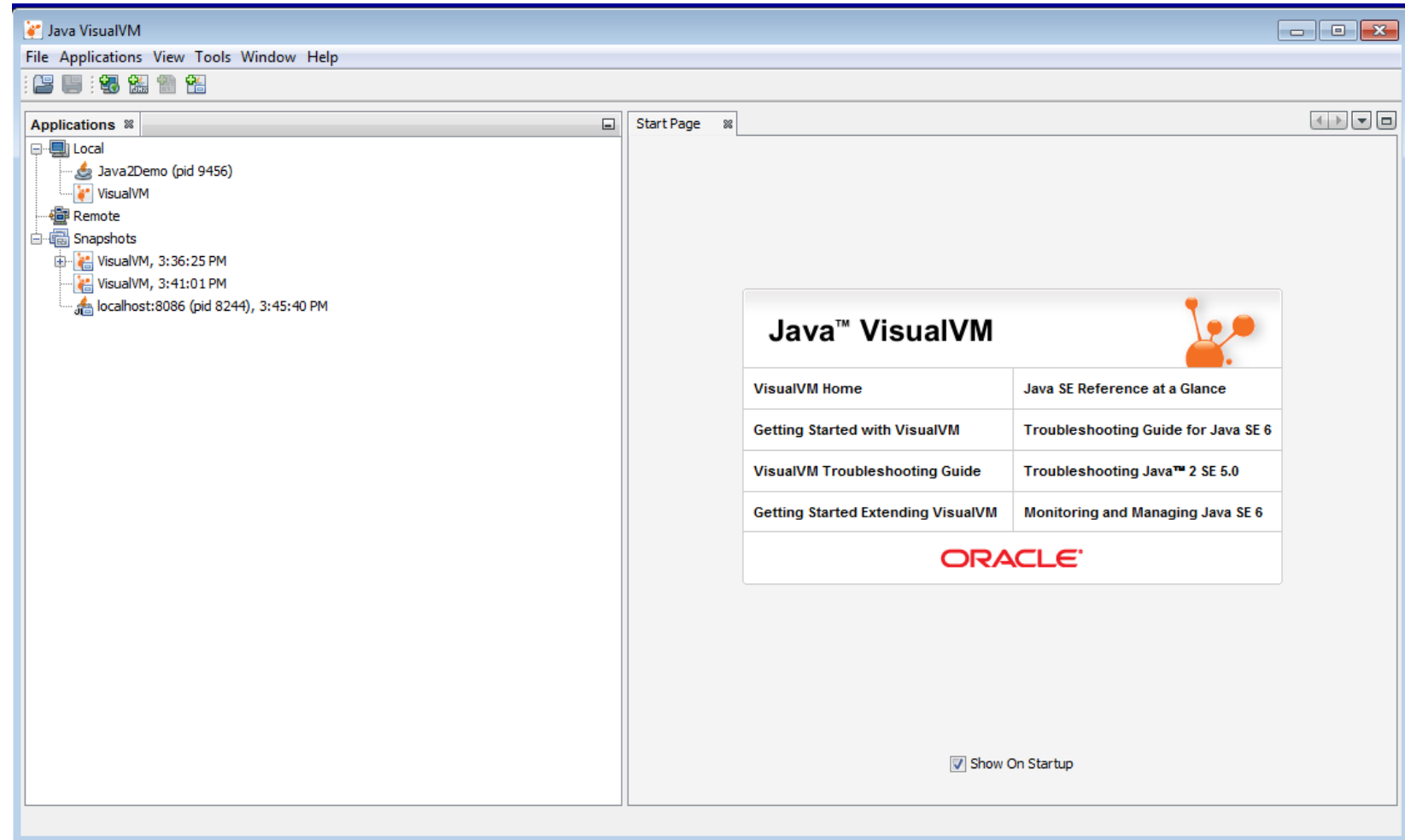
## Start Transforms





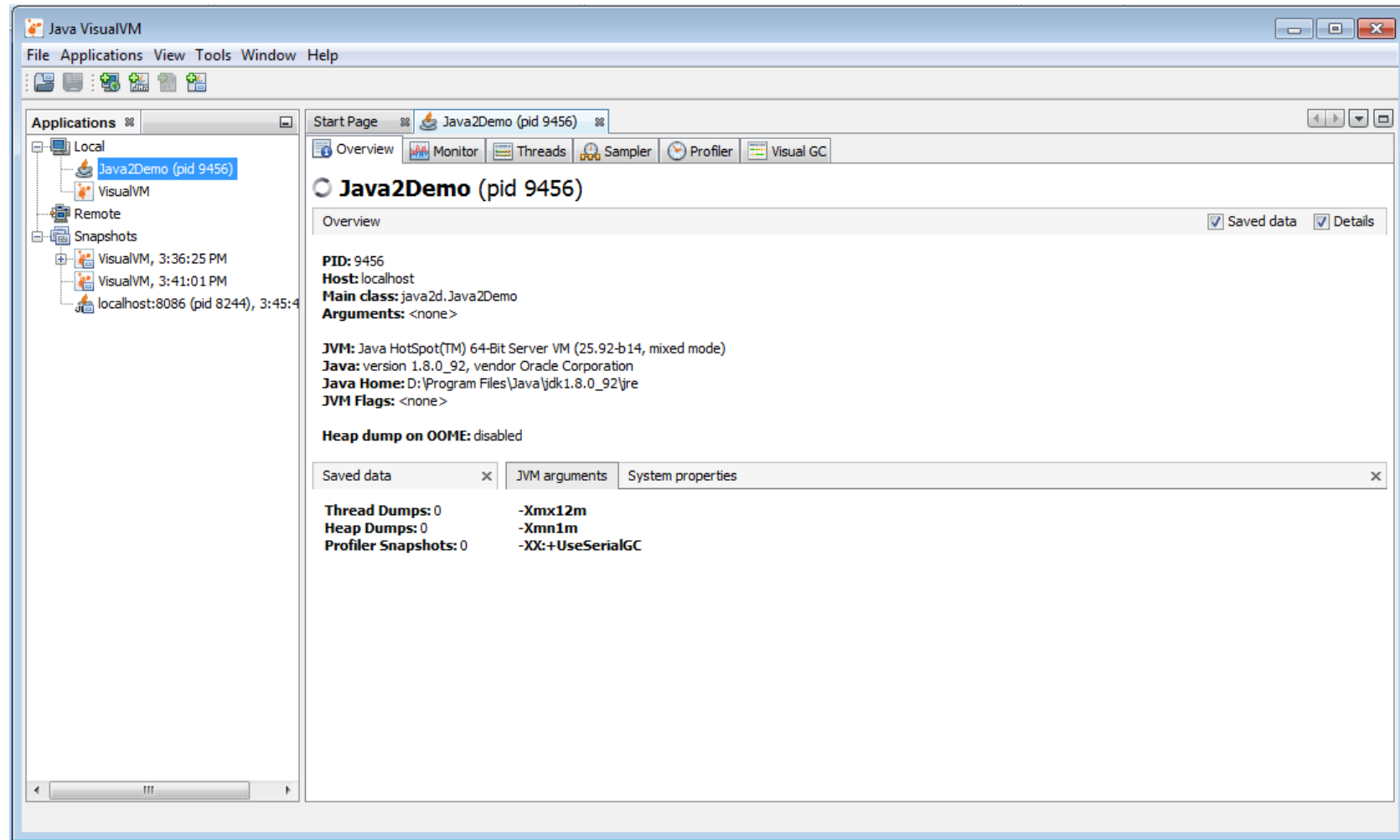
# Garbage Collection – Demo Application

## Start VisualVM



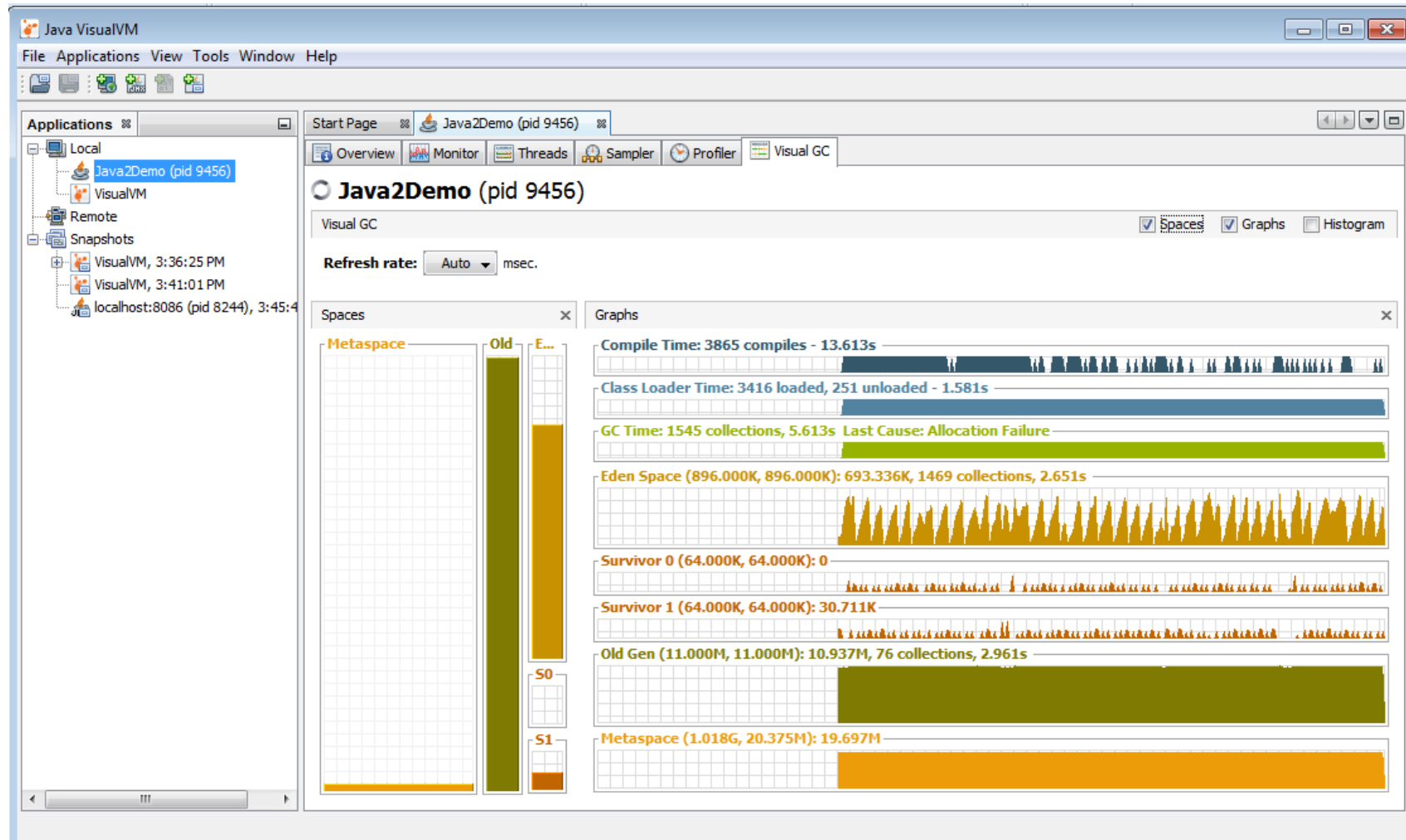
# Garbage Collection – Demo Application

Start Java2D demo application to visualize GC activities



# Garbage Collection – Demo Application

## Visual GC



# Generational Garbage Collection Types

## Java Garbage Collectors

You now know the basics of garbage collection and have observed the garbage collector in action on a sample application. In this section, you will learn about the garbage collectors available for Java and the command line switches you need to select them.

Switch	Description
-Xms	Sets the initial heap size for when the JVM starts.
-Xmx	Sets the maximum heap size.

# Generational Garbage Collection Types

## Java Garbage Collectors – Serial GC

The serial collector is the default for client style machines in Java SE 5 and 6. With the serial collector, both minor and major garbage collections are done serially

### Usage cases

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines.

# Generational Garbage Collection Types

## Java Garbage Collectors – Serial GC

### Command Line Options:

To enable the Serial Collector use:  
-XX:+UseSerialGC

```
java -Xmx12m -Xms3m -Xmn1m -XX:+UseSerialGC -jar  
c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

# Generational Garbage Collection Types

## Java Garbage Collectors – Parallel GC (throughput collector)

The parallel garbage collector uses multiple threads to perform the young generation garbage collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection.

### Usage cases

It can use multiple CPUs to speed up application throughput. This collector should be used when a lot of work needs to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

# Generational Garbage Collection Types

## Java Garbage Collectors – Parallel GC

### Command Line Options:

To enable the Parallel Collector use:

`-XX:+UseParallelGC`

The number of garbage collector threads can be controlled with command-line options:

`-XX:ParallelGCThreads=<desired number>`



# Generational Garbage Collection Type

## Java Garbage Collectors – CMS GC (Concurrent Mark Sweep)

It collects the tenured generation. It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads. Normally the concurrent low pause collector does not copy or compact the live objects. A garbage collection is done without moving the live objects. If fragmentation becomes a problem, allocate a larger heap.

### Usage cases

The CMS collector should be used for applications that require low pause times and can share resources with the garbage collector. Examples include desktop UI application that respond to events.

# Generational Garbage Collection Types

## Java Garbage Collectors – CMS GC

### Command Line Options:

To enable the CMS Collector use:  
`-XX:+UseConcMarkSweepGC`

and to set the number of threads use:  
`-XX:ParallelCMSThreads=<n>.`

# Generational Garbage Collection Types

## Java Garbage Collectors – G1 GC (Generation 1)

The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector that has quite a different layout from the other garbage collectors described previously

### Command Line Options:

To enable the G1 Collector use:

To enable the G1 Collector use:  
`-XX:+UseG1GC`

# Thanks