Android OS - Memory Management in Android

Java has automatic memory management. It performs **routine garbage collection** to clean up unused objects and free up the memory. **However**, it is very important for us to know how the **garbage collector** works in order to manage the application's memory effectively. Thus avoiding **OutOfMemoryError** and/or **StackOverflowError** exceptions.

Memory structure

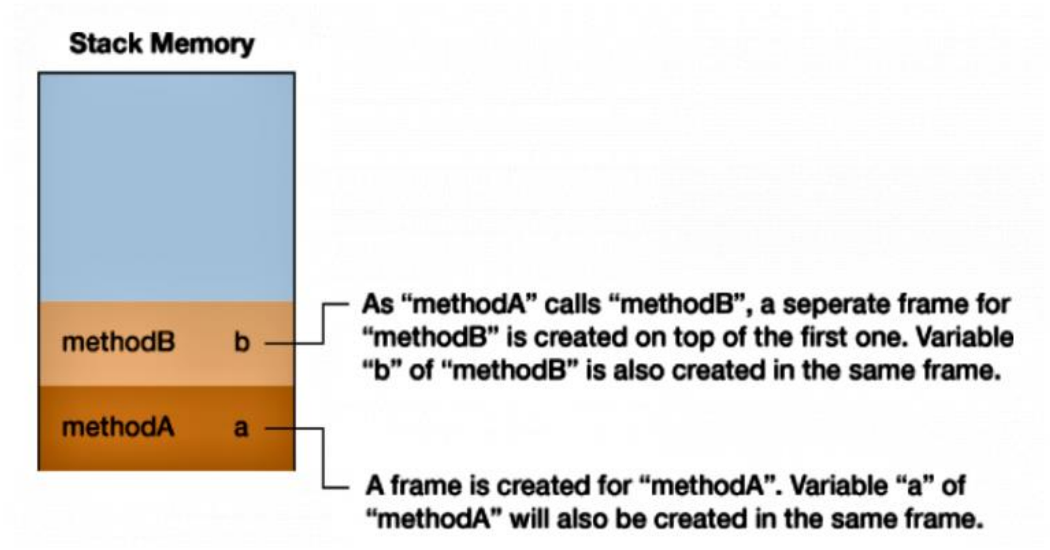
For effective memory management, **JVM** divides memory into **Stack** and **Heap**.

1. Stack Memory

Java Stack memory is used for the **execution of the thread**. They contain **method-specific values** which that are **short-lived** and **references** to the other objects in the **heap** that are getting referred from the **method**.

Example:

```
public void methodA() {
      int a = 10;
      methodB(a);
}
public void methodB(int value) {
      int b = 10;
      //Rest of the code.
}
```


Stack Memory
methodB b
methodA a
As "methodA" calls "methodB", a seperate frame for "methodB" is created on top of the first one. Variable "b" of "methodB" is also created in the same frame.
A frame is created for "methodA". Variable "a" of "methodA" will also be created in the same frame.

From the above picture:

- Variable "**b**" of "**methodB**" can be accessed by "**methodB**" only and not by "**methodA**", as "**methodA**" is in separate frame.
- Once the "**methodB**" execution is **completed**, the **control** will go to the calling function. In this case, it's "**methodA**".

- Thus, the frame for "**methodB**" will be removed from the stack and all the variables in that frame will also be flushed out. **Likewise**, for "**methodA**".

2. Heap Memory

Java heap **space** is used to **allocate memory to the objects**. Whenever we create Java/Kotlin objects, these will be **allocated** in the **Heap** memory. **Garbage collection process** runs in the heap memory.

Garbage Collection Process

- Garbage Collection is a **process** of cleaning up the **heap memory**.
- Garbage collector **identifies** the **unreferenced objects** and removes them to free the memory space.
- The objects that are being referenced are called '**Live objects**' and those which are not referenced are called **'Dead objects'**.

Process involved in Garbage collection.

Step 1: Marking

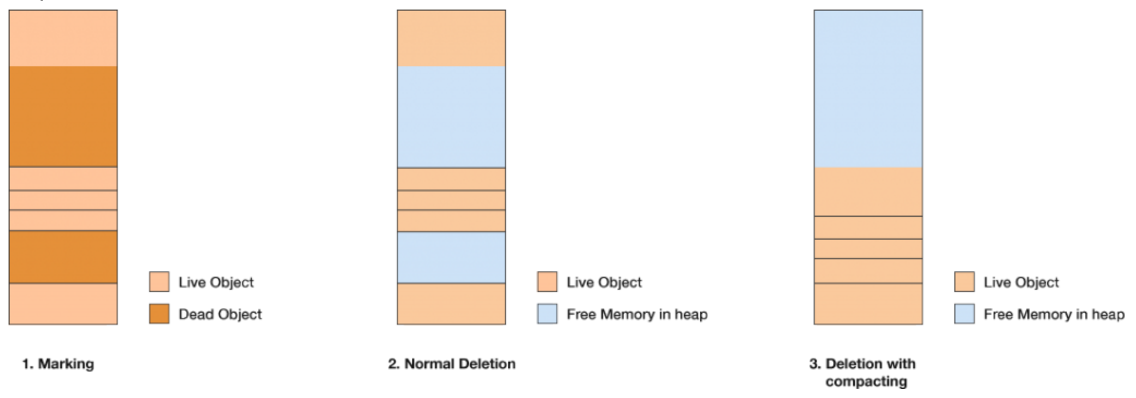
- Most of us think that Garbage Collector marks dead objects and removes them.
- **In reality**, it is exactly the opposite. Garbage Collector first finds the **'Live objects'** and marks them. This means the rest of the objects that are not marked are '**Dead objects'.**

Step 2: Normal Deletion

- Once Garbage Collector finds the **'Dead objects'**, it will **remove** them from the **memory**.

Step 3: Deletion with Compacting

- **Memory allocator** holds the reference of the **free memory space** and searches for the same whenever new memory has to be allocated.
- In order to improve performance, it is **better** if we move all the referenced objects to one place.



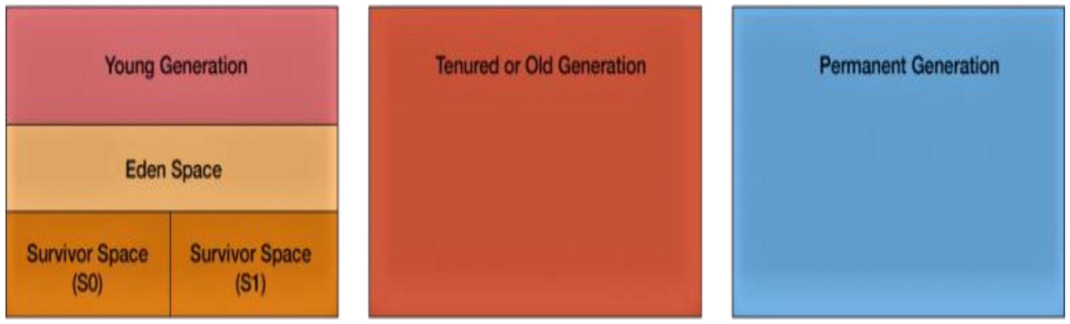
Basic GC process

This algorithm is called a mark-sweep-compact algorithm.

As the number of objects increase, the above process i.e., Marking, Deletion and Deletion with compacting is inefficient. As per the empirical analysis, most objects are short-lived. Based on this analysis, the heap structure is divided into three generations**.**

Heap Structure

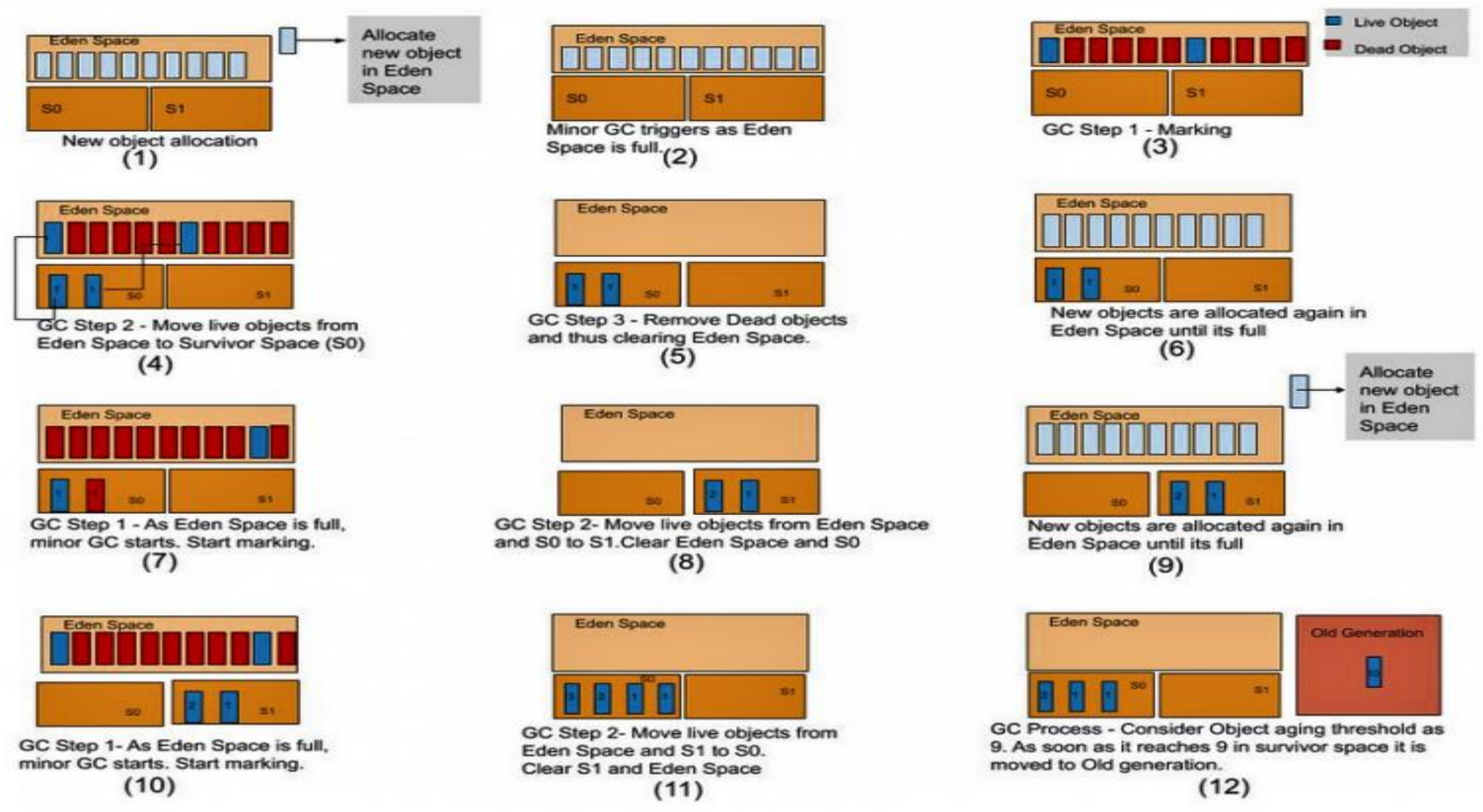
The heap structure is divided into three divisions, **Young** Generation, **Tenured** or **Old** Generation, and **Permanent** Generation.



Heap Structure

1. **Young Generation –** This is where **all the new objects are allocated and aged**. This **generation** is **split** into **Eden** Space and **Survivor** spaces.
2. **Eden Space – All new objects are allocated here**. Once this space is full, minor Garbage Collection will be triggered.
3. **Survivor Space –** After Minor GC, the live objects from **Eden space** will be **moved** to one of the survivor spaces **S0** or **S1**.

1. <span>**Young Generation :**</span>the below diagram describes the Garbage Collection process in Young Generation.



Eden Space
S0
S1
Allocate new object in Eden Space
New object allocation
(1)
Minor GC triggers as Eden Space is full.
(2)
Live Object
Dead Object
GC Step 1 - Marking
(3)
GC Step 2 - Move live objects from Eden Space to Survivor Space (S0)
(4)
GC Step 3 - Remove Dead objects and thus clearing Eden Space.
(5)
New objects are allocated again in Eden Space until its full
(6)
GC Step 1 - As Eden Space is full, minor GC starts. Start marking.
(7)
GC Step 2- Move live objects from Eden Space and S0 to S1.Clear Eden Space and S0
(8)
New objects are allocated again in Eden Space until its full
(9)
GC Step 1- As Eden Space is full, minor GC starts. Start marking.
(10)
GC Step 2- Move live objects from Eden Space and S1 to S0. Clear S1 and Eden Space
(11)
Old Generation
GC Process - Consider Object aging threshold as 9. As soon as it reaches 9 in survivor space it is moved to Old generation.
(12)

GC process in Young Generation

Note: at any given time **only one survivor space has objects**. Also, note that **the age of the object keeps increasing** when switching between the survivor spaces.

2. **Old Generation –** long-surviving objects will be stored. As mentioned, a threshold will be set to the object, on meeting which it is moved from the young generation to old or tenured generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.
3. **Permanent generation –** This contains **metadata** required by the **JVM** to **describe** the **classes** and **methods** used in the application.

- **Improvements**: In modern JVMs (from **Java 8** onwards), the **Permanent Generation** has been **replaced** with **Metaspace** to better handle dynamic memory requirements.
- **Metaspace:** is not of fixed size and expands dynamically based on the application's demand for loading **classes** and **metadata**. It **resides** in **native memory** rather than in the **JVM heap**, which improves flexibility and reduces the likelihood of **PermGen-related** errors.

Types of Garbage Collectors

- Serial GC
- Parallel GC
- **Concurrent Mark and Sweep (CMS) collector**
- G1 Collector

These garbage collectors have their own **advantages** and **disadvantages**. As **Android Runtime** (**ART**) uses the concept of **CMS** collector.

GC Algorithms

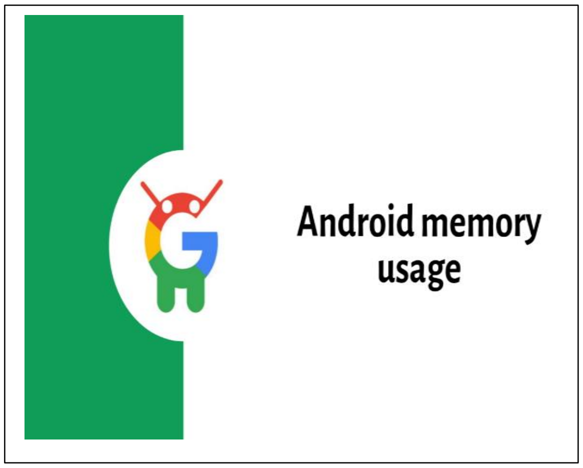
- usually **two** different GC algorithms are needed
- **One** for the **Young generation** and the **other** for the **Old generation**.
- The default **GC** type used by **ART** is **CMS** Collector.

Concurrent Mark & Sweep (CMS) Collector

- This collector is used to avoid long pauses during the Garbage collection process.
- It scans heap memory using multiple threads.
- It uses parallel Stop the World mark-copy algorithm in the **young generation**.
- It uses concurrent mark-sweep algorithm in the **Old Generation**.
- **Minor GC** occurs in **young generation** whenever **Eden Space** is full. And this is "Stop the World event".
- GC process in **Old generation** is called **Major GC**.

Understanding Memory Usage in Android

The whole lecture can be divided into five parts:



1. Why care about memory usage?
2. How memory use impacts a device?
3. Evaluating application memory impact.
4. Reducing your application's memory impact.
5. Closing notes.

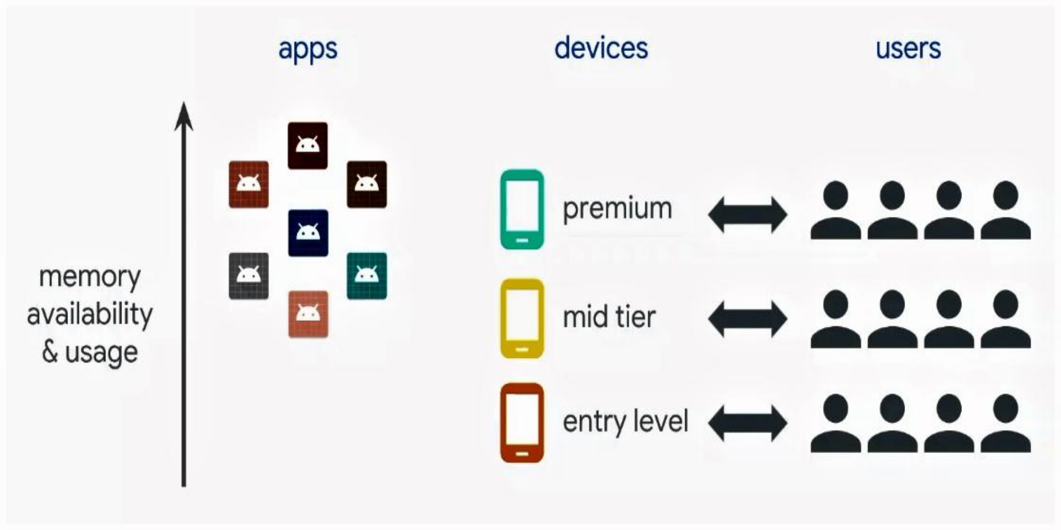
Why care about memory usage?

Basically, there are three types of Android devices i.e.

1. Entry-level devices
2. Mid-tier devices
3. Premium devices

Out of these three types of devices, the **entry-level devices** have **low memory** and it can run only those applications that **require low memory**.

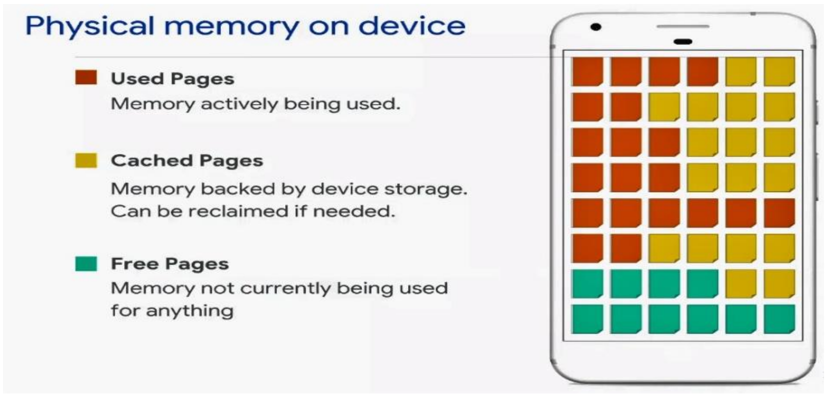
So, if the memory usage of application **increases** then it will be difficult for the **entry-level devices** to run that application and **as a result**, you will lose your users. This will affect the Android **ecosystem** (i.e. a collection of apps, devices, and users)



NOTE: see video from **1:00** to **1:48** https://youtu.be/w7K0jio8afM?t=60

How memory use impacts a device?

- The memory of the Android device is divided into **pages** and each **page** is around **4 kilobytes. There are three types of pages**:

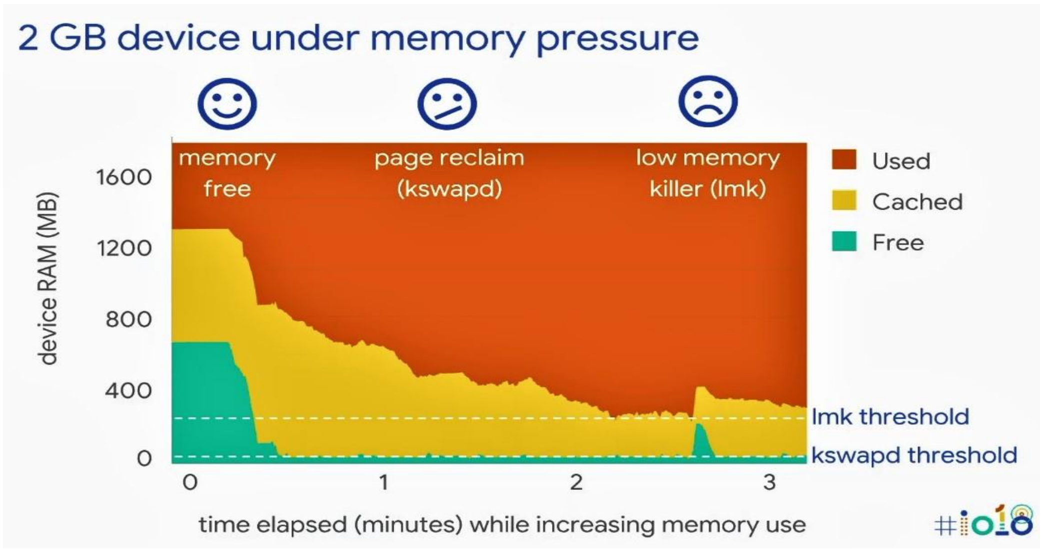

Physical memory on device
Used Pages
Memory actively being used.
Cached Pages
Memory backed by device storage. Can be reclaimed if needed.
Free Pages
Memory not currently being used for anything

1. **Used Pages:** These are the **pages** that are **currently being used** by the **processes**.

2. **Cached Pages:** These are the pages that the **processes** are using **but** some part of the memory is also present in the main memory. So, to have a **fast retrieval** of data, we use cached pages.

3. **Free Pages:** These are the pages that are free i.e. these are the memory space that can be used to store something in the future.

The following graph shows the memory usage during the course of time.
In the **beginning**, there is nothing to run but with the due course of time, we started using more applications and this, in turn, use more and more memory over time.



From the above graph

- We can see that in the **beginning**, when the device started running then there is a lot of free memory available.
- But when we start using other applications then the free memory is being used.
- To avoid something bad that can happen due to low memory, the **kernel** performs an operation called **kswapd.**

> - **kswapd** stands for "**kernel swap daemon**," which is a component of the **Linux** operating system kernel. **kswapd** is a background process responsible for **managing virtual memory and swap space**.
> - Its primary role is **to ensure that there is enough physical memory (RAM) available** for running applications and processes.
> - **kswapd** achieves this by freeing up unused memory pages or moving them to swap space when necessary.

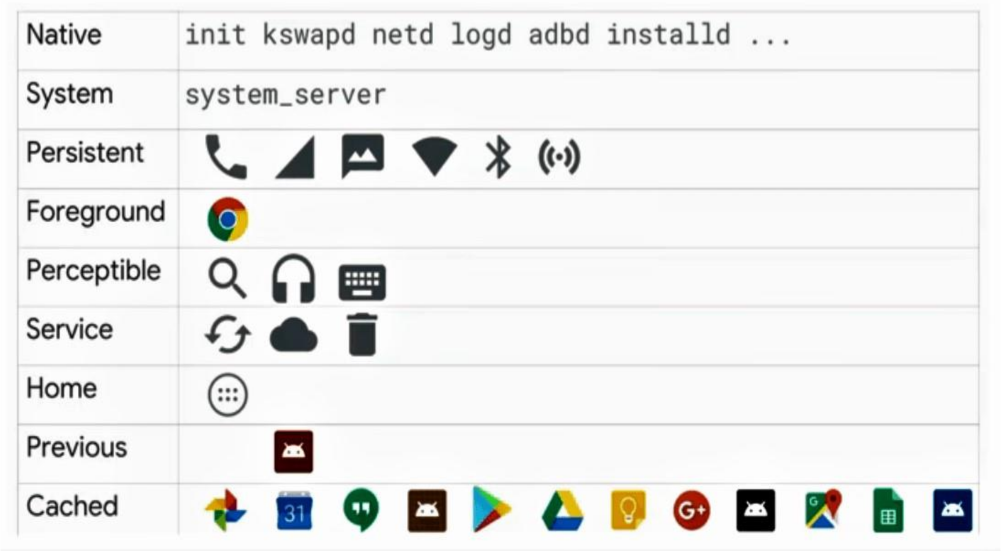
➢ In the process of **kswapd** , if the memory of the device goes down the **kswapd threshold** , then the Linux kernel starts finding some more free memory. So, what is does is, it **reclaims the cached pages** and make it free. **But the problem** here is that if an app wants to reclaim the memory present in the cached pages then it will take some time because at present there is nothing in the cached page and data will be loaded from the device storage.

➢ But what if we continue using more and more application. **In this case**, due to the process of **kswapd** , more cache memory will be cleared and a time will come when the device starts to thrash. And this is a very bad thing because the device will be completely locked up.

➢ In Android, we have a process called **low memory killer,** and this will pick a process from the device and will completely kill that process. By doing so, you will get back all the memory that the process was using. But what if the **low memory killer,** kills the process that the user cares about?

➢ In Android, we have a priority list of applications and based on that priority list we remove the app when the **low memory killer** comes into play.

➢ **Following is the priority list in Android:**

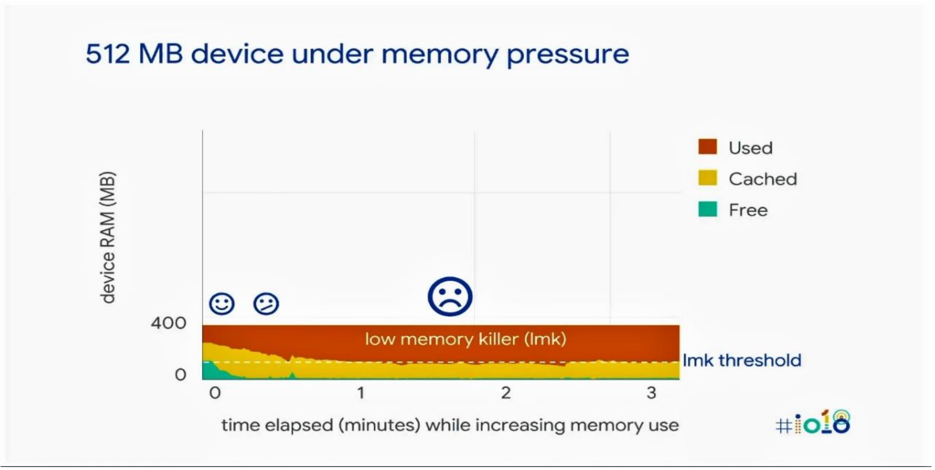


Native	`init kswapd netd logd adbd installd ...`
System	`system_server`
Persistent	📞 ◢ 🖼 ▼ ✱ (•)
Foreground	●
Perceptible	🔍 🎧 ⌨
Service	⟳ ☁ 🗑
Home	⊙
Previous	■
Cached	■ 31 ● ■ ▶ ▲ ■ G+ ■ ■ ■ ■

➢ Whenever the low memory killer comes into play, **Then**

- It first deletes the cached applications.
- Even after that, the memory usage keeps on increasing then the **low memory killer kills** the previously opened applications.
- Again if the memory usage keeps on increasing, then the home application will be **killed** and after that service, perceptible, foreground and persistent apps will be stopped or killed.
- Again, if the memory usage keeps on increasing, then the system app will be stopped and your phone will be **rebooted.** **:(** This is the worst user experience that we can have in low memory devices or entry-level devices.

The example that we took was a **2GB** device. The situation becomes worse when you are having a **512MB** device.

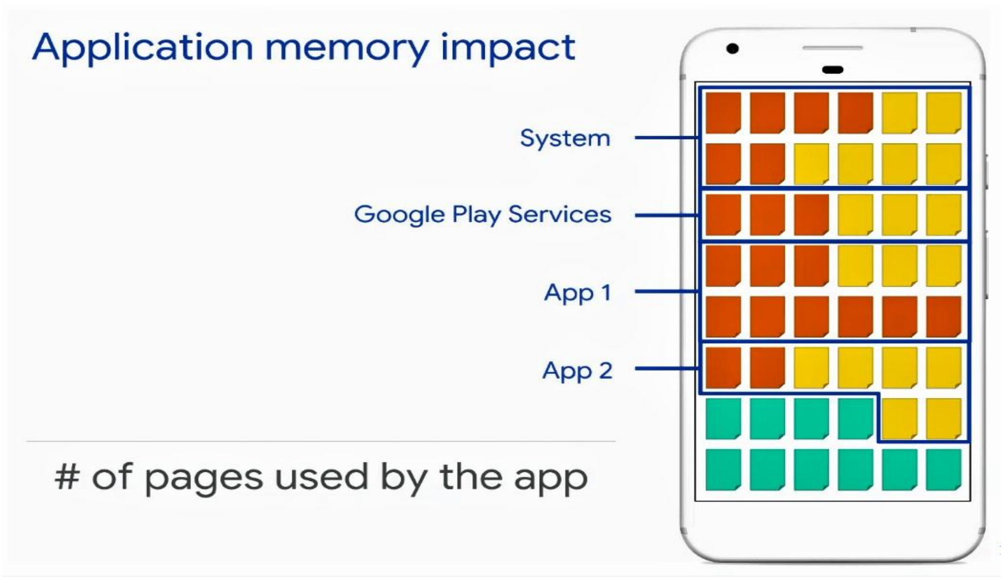


In the above example, we have very less memory and due to this the **kswapd** and **low memory killer** will come into play very earlier. So, **low memory killer** will always be active and this will result in bad user experience and we should try our best to develop an application which can perform well in low memory situations also.

NOTE: see video from **2:30** to **12:20** https://youtu.be/w7K0jio8afM?t=150

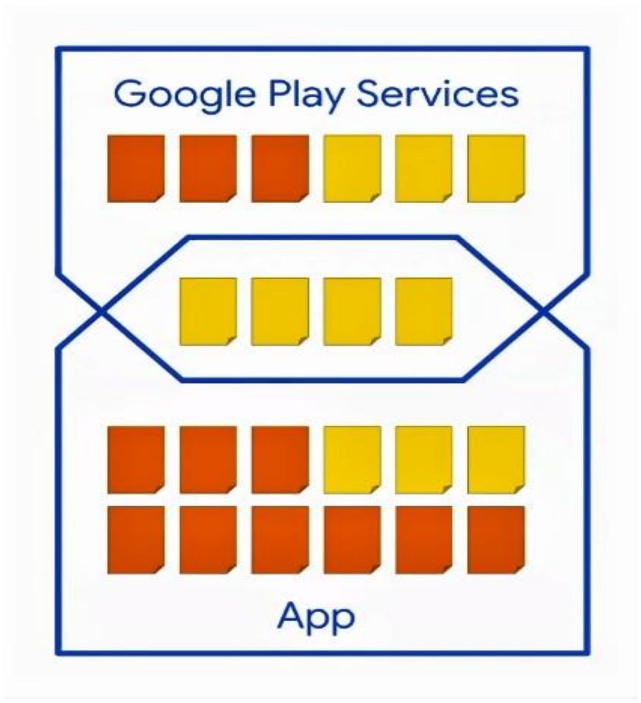
Evaluating application memory impact

In the above section, we have seen that the memory of the device is divided into pages and the Linux kernel system keep a track of the pages used by a certain application.



But the situation gets worse when the application starts using some **shared memory**.

For example, you can have an application that calls the **Google Play Services** and this, in turn, result in page sharing between these two applications.



So, the problem that arises here is how to deal with this shared memory.

There are a few different ways that can be used to deal with these situations:

1. **RSS (Resident Set Size):** In this method, the **application** is **responsible** for all the shared memory.
2. **USS (Unique Shared Set):** the **application** is **not responsible** for any of the shared pages.
3. **PSS (Proportional Set Size):** the **app** will be responsible for that number of pages that are proportional to the number of processes sharing the shared memory. **For example**, if the **shared memory** contains 4 pages and the memory is shared among **two processes** then the app will be responsible for only two pages. If the shared **page** is **4** and the number of sharing processes will be 3 then the app will handle **4/3 pages**.

So, which method to choose from the above?

- If the shared memory is being used by the **application** only then we should use the **RSS** approach.
- If the shared memory is taken by the **Google Play** Services then we should use the **USS** approach.

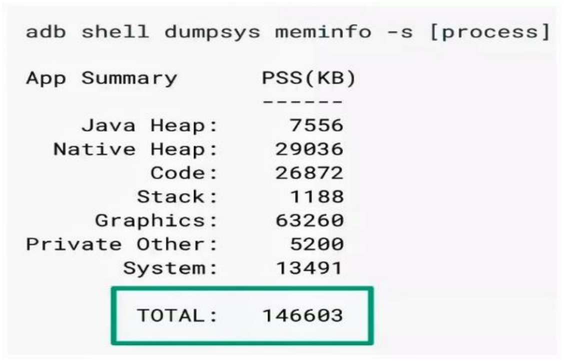
But in general, it is very difficult to find if the app will require shared memory or not. So, we use the **Proportional Set Size** method. **PSS** avoids over-counting or under-counting the overall impact of shared pages on a device.

So, we can use **PSS** to find your application's memory impact and to find the **PSS** value of your application, you can run the below command:

```
abd shell dumpsys meminfo -s [process]
```

Here, your process name can be ***com.example.android.example*** .

So, when you run the above command then you will get something like below and the total value here is the **PSS** value of your process.

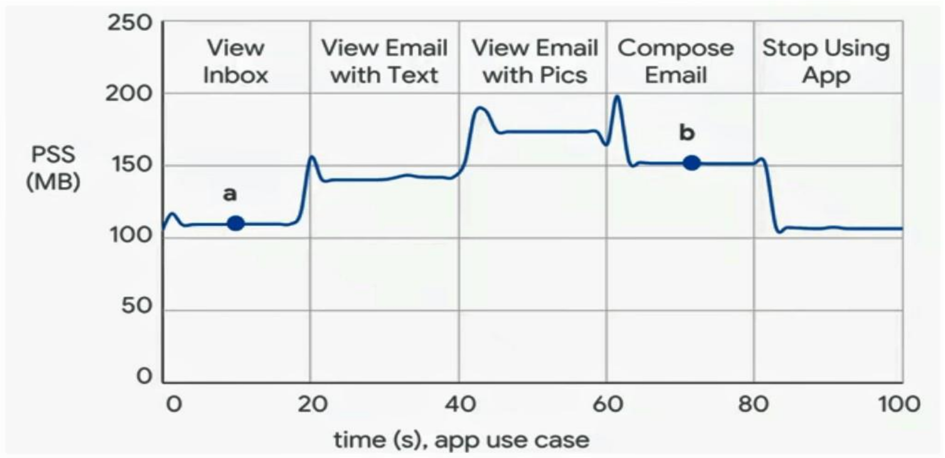


```
adb shell dumpsys meminfo -s [process]

App Summary        PSS(KB)
                   ------
    Java Heap:       7556
  Native Heap:      29036
         Code:      26872
        Stack:       1188
     Graphics:      63260
Private Other:       5200
       System:      13491

        TOTAL:     146603
```

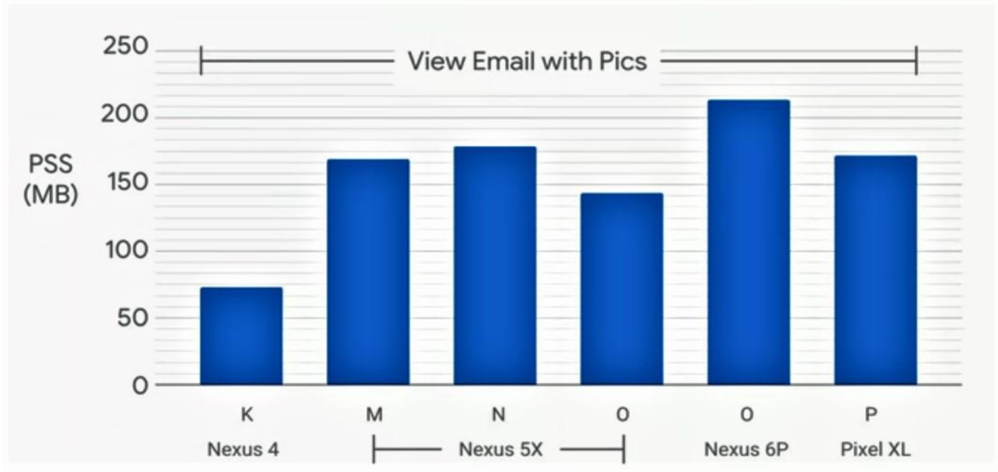
The point that needs to be noted here is that there are various other factors that are responsible for the performance of the app. **They are:**

1. The application use case
2. The platform configuration
3. The device memory pressure

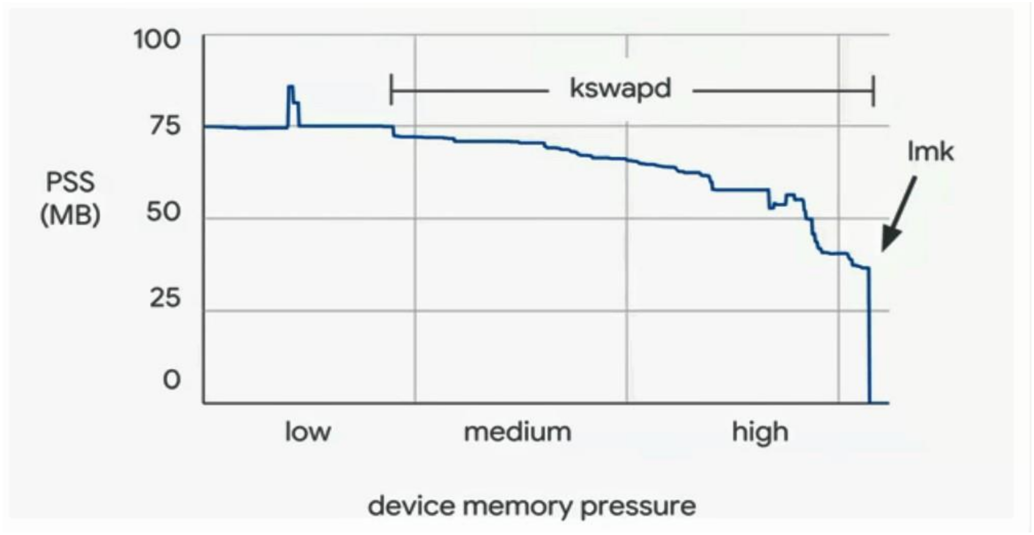
- You shouldn't only depend on the **PSS** value. But you should also care about the application **use-case**. **For example**, in the following graph, there are various **use-cases** of the **Gmail application**. You can see that you can't compare the **PSS** value of point **a** with point **b** because these points are in different **use-case.**



- **Also**, your device performance will be dependent on **platform configuration** i.e. the **premium** device can perform more in comparison with the **entry-level** devices. Also, it depends on the **Android version**. So, when you are testing an application, then you should use a **particular device** and a **particular Android version**.



- **Lastly**, the impact of memory depends on **memory pressure**.
  For example, in the below figure, we stared the chrome application and since there is no memory pressure the **PSS** value is constant but eventually when the pressure on the device increases that is more and more applications are in use, then **kswapd** comes into role and some of the cached pages will be killed and there will be a decrease in the **PSS** value.

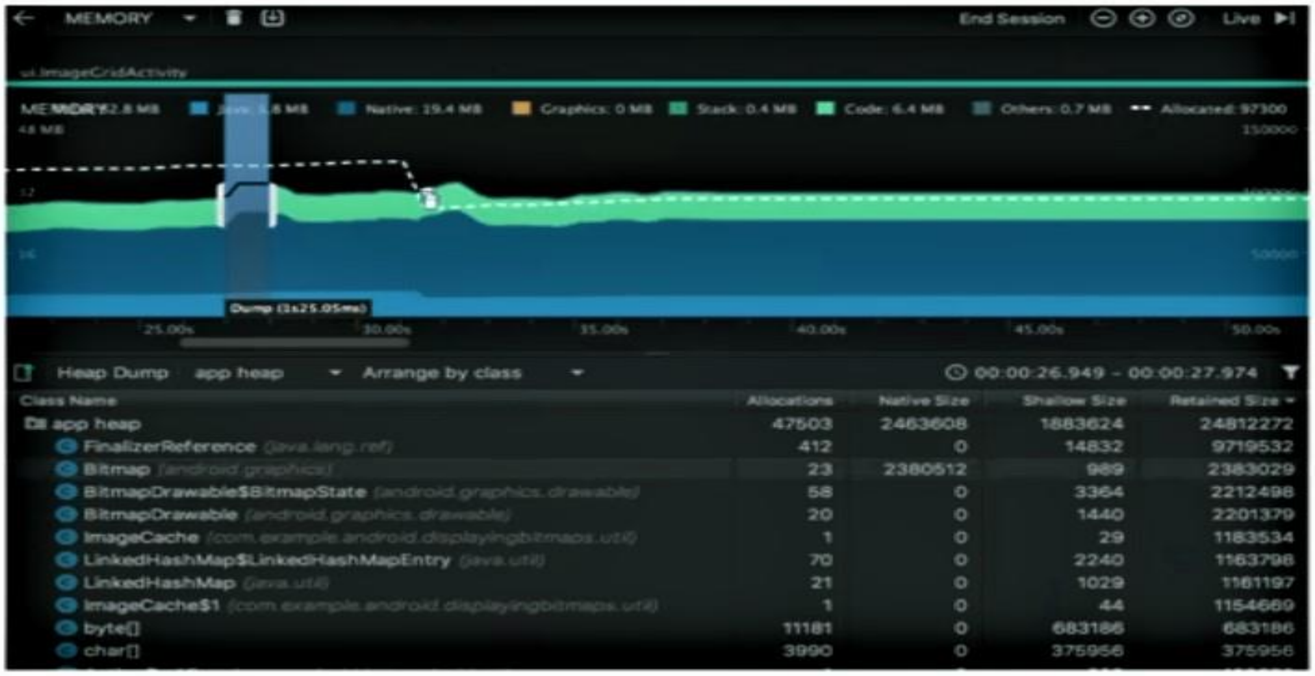


- So, it is advised to use some devices that are having **enough memory (RAM)** so that the high-pressure memory doesn't come into play.

NOTE: see video from **12:21 to 26:08** https://youtu.be/w7K0jio8afM?t=741

Reducing your application’s memory impact

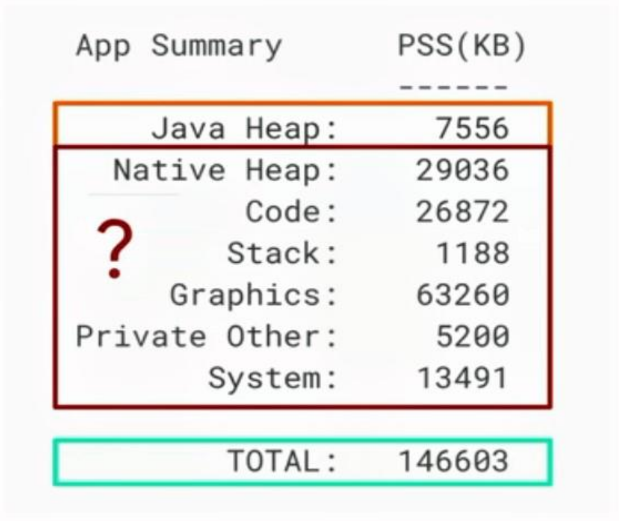
The very first tip to reduce your application’s memory impact can be checking the Android Studio’s **memory profiler**.



Class Name	Allocations	Native Size	Shallow Size	Retained Size
app heap	47503	2463608	1883624	24812272
FinalizerReference (java.lang.ref)	412	0	14832	9719532
Bitmap (android.graphics)	23	2380512	989	2383029
BitmapDrawable$BitmapState (android.graphics.drawable)	58	0	3364	2212498
BitmapDrawable (android.graphics.drawable)	20	0	1440	2201379
ImageCache (com.example.android.displayingbitmaps.util)	1	0	29	1183534
LinkedHashMap$LinkedHashMapEntry (java.util)	70	0	2240	1163798
LinkedHashMap (java.util)	21	0	1029	1161197
ImageCache$1 (com.example.android.displayingbitmaps.util)	1	0	44	1154669
byte[]	11181	0	683186	683186
char[]	3990	0	375956	375956

This will give you a ton of information i.e. you will come to know about your **java object**, where are they **allocated**, what’s holding on to them and almost everything that you can find about your Java object.

While finding the PSS value, we only considered the **Java heap**. But what about other factors that are responsible for reducing your application’s memory impact.



App Summary	PSS(KB)
Java Heap:	7556
Native Heap:	29036
Code:	26872
Stack:	1188
Graphics:	63260
Private Other:	5200
System:	13491
TOTAL:	146603

These things depend on the Android platform. To get full information of the **PSS** value, you can use ***-a*** in the ***dumpsys*** command:

```
dumpsys meminfo -a
```

dumpsys is a tool that runs on Android devices and provides information about system services. Call **dumpsys** from the command line using the **Android Debug Bridge (ADB)** to get diagnostic output for all system services running on a connected device.

This will give a much more detailed breakdown of the **PSS** value. It will also show you the **breakdown** of different categories of memory like private, clean, share, dirty and so on.

If the above information is not useful then you can use ***showmap*** in your application and this will give more breakdown of your **memory mapping**.

These platform commands can be used but the problem here is that:

1. These tools aren't well supported
2. The user interface is clumsy
3. You need deep platform expertise
4. May need a rooted device

So, the idea here is that the information shown to you is not that useful or you need to be an expert to understand those reports. So, this is not going to be a good tool.

What to do next? Is there any way?

Yes, if you want to improve your overall **memory** use then you can do **two** things:

1. **Profile your Java heap:** You can profile your Java heap with the help of Android Studio Profiler. Since the allocations that are outside the **Java heap** are tied to Java allocation. So, your **application** will be **calling** the Android framework, the **Android framework** is **calling** into the native libraries and this is doing native allocations. The lifetime of these will be tied to Java objects. So, **just profile your Java heap**.

2. **Reduce your APK size:** You can reduce the **APK** size of your application because there are a lot of things that are present in **APK** and this, in turn, affects the runtime. So, try to reduce the **APK** size of your application. To reduce the **APK** size, you can visit the Android developer website

Closing Notes

In this lecture,

- We understand Android Memory usage.
- We saw how the memory use can impact a device and at last, we saw how to reduce your application's memory impact.
- One thing that can be noted here is that you have to compromise between memory usage and user satisfaction because if your application is using very low memory then the features of the app will be less. Similarly, when the features are high then the memory use is also high.
- Also, with due respect of time, the size of the app increases i.e. the app size in **2017** will not be same in **2019**. So, these things also come into play.

NOTE: see video from **26:10 to End** https://youtu.be/w7K0jio8afM?t=1570