Provera koriscenja memorije od strane aplikacije uz pomoc Memory Profiler-a

Sadrzaj

[**Uvod** 2](#_Toc127974626)

[**Zasto treba profilisati koriscenje memorije od strane aplikacije** 2](#_Toc127974627)

[**Pregled Memory Profiler-a** 3](#_Toc127974628)

[Kako se memorija racuna 3](#_Toc127974629)

[**Pregled memorijskih alokacija** 4](#_Toc127974630)

[Pregled globalnih JNI reference 7](#_Toc127974631)

[Native Memory Profiler 7](#_Toc127974632)

[**Snimanje heap dump-a** 8](#_Toc127974633)

[Cuvanje heap dump-a u HPROF fajlu 10](#_Toc127974634)

[**Detekcija curenja memorije u Memory Profiler-u** 10](#_Toc127974635)

# **Uvod**

**Memory Profiler** je komponenta unutar Android Profiler alata koji nam pomaze da identifikujemo memory leak-ove koji mogu dovesti do nepravilnog rada aplikacije, zamrzavanja I zakucavanja. Prikazuje realtime grafikon koriscenja memorije od strane aplikacije I pomaze nam da snimimo heap dump, da aktiviramo garbage collector, I da pratimo alokaciju memorije.

Da bismo otvorili Memory Profiler u Android Studio-ju, treba ispratiti sledece korake:

* Kliknuti na **View > Tool Windows > Profiler**
* Odabrati uredjaj I process aplikacije koju zelimo da profilisemo uz pomoc Profiler-a.
* Kliknuti bilo gde na **Memory timeline** da bi otvorili Memory Profiler.

# **Zasto treba profilisati koriscenje memorije od strane aplikacije**

Android nudi okruzenje za upravljanje memorijom (**managed memory environment**) koji odlucuje kada aplikacija ne koristi odredjene objekte u memoriji I poziva garbage collector oslobadja nekoriscenu memoriju nazad heap-u. Android stalno unapredjuje nacine za pronalazak nekoriscenih memorijskih lokacija, ali kod svih verzija Androida sistem mora nakratko pauzirati izvrsavanja naseg koda. U vecini vremena, te pauze su neprimetne. Medjutim, ako aplikacija dodeljuje memoriju brze nego sto sistem moze da je prikupi, mozemo doci do situacije da aplikacija bude pauzirana vise nego sto je “ocekivano” posto ceka da garbage collector oslobodi dovoljno memorije da zadovolji potrebno alociranje. To cekanje izaziva kasnjenje koje moze dovesti do toga da aplikacija preskace frame-ove I izazove vizualno usporavanje aplikacije.

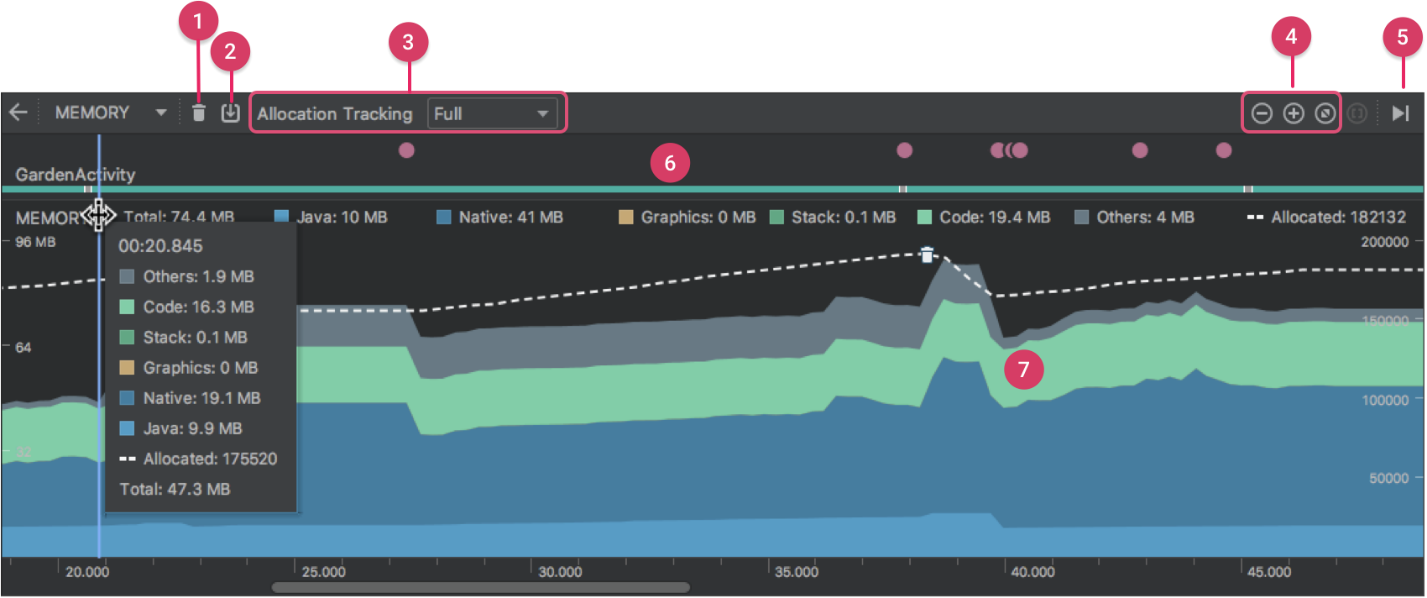
Cak iako aplikacija ne pokazuje tu sporost I to kasnjenje, ukoliko dodje do memory leak-a aplikacija ga moze zadrzati cak iako je u background-u. Ovo ponasanje moze usporiti perfomanse kompletnog sistema time sto ce inicirati nepotreban rad garbage collectora. Eventualno, sistem ce na silu ubiti process aplikacije da bi preuzeo memoriju koja mu je potrebna za bitnije procese (sistemske potencijalno). Ukoliko je ona u background-u, I korisnik se vrati na aplikaciju, ona ce se restartovati kompletno.

Da bismo izbegli ove probleme, mozemo koristiti Memory Profiler za sledece akcije:

* Trazenje nezeljene obrasce dodeljivanje memorije u timelinu koji mogu dovesti do problema u perfomansi aplikacije.
* Mozemo dump-ovati Java heap da vidimo koje objekti koriste memoriju u bilo kom vremenskom interval. Nekoliko heap dump-ova u toku duzeg vremenskog perioda moze dovesti do oktrivanja uzroka curenja memorije (memory leak).
* Snimanje alokacije memorije tokom normalne ili ekstremne korisnicke interakcije da bismo tacno identifikovali gde kod ili dodeljuje previse objekata u kratkom vremenu ili dodeljuje objekte koji ce biti leak-ovi.

# **Pregled Memory Profiler-a**

Kada otvorimo Memory Profiler, mozemo videte detaljan timeline koriscenja memorije I alate koji mogu da “force”-uju garbage collector da ocisti memoriju, hvatanje heap dump-ovanja I pregled memorijskog alociranja.



1. Dugme za force-ovanje garbage collection event
2. Dugme za “capture heap dump”
3. Meni za specifikaciju ucestalosti capture-ovanja memorijskih alokacija. Selektovanje odgovarajuce opcije moze nam pomoci da unapredimo perfomanse aplikacije tokom profilisanja
4. Dugmad za zumiranje timeline-a
5. Dugme za prelazak unapred na memorijske podatke uzivo
6. Event timeline, koji pokazuje stanja activity-ja, korisnicke evente I promene rotacije ekrana
7. Timeline koriscenja memorije, koji ukljucuje sledece:
   1. Naslagani grafikon koji pokazuje koliko memorije koristi svaka memorijska kategorija, sto je oznaceno y osom sa leve strane I tasterom u boji na vrhu
   2. Isprekidana lonija oznacava broj dodeljenih objekata kao sto je oznaceno y osom sa desne strane
   3. Ikonica za svaki garbage collection event-a.

## Kako se memorija racuna

Brojevi koji se vide na vrhu Memory Profiler-a se baziraju na kompletnoj memoriji I katagorijama koje aplikacije koristi. Ovde ne spadaju stranice koje se dele sa sistemom ili drugim aplikacijama.



Kategorije koje se nalaze ovde su:

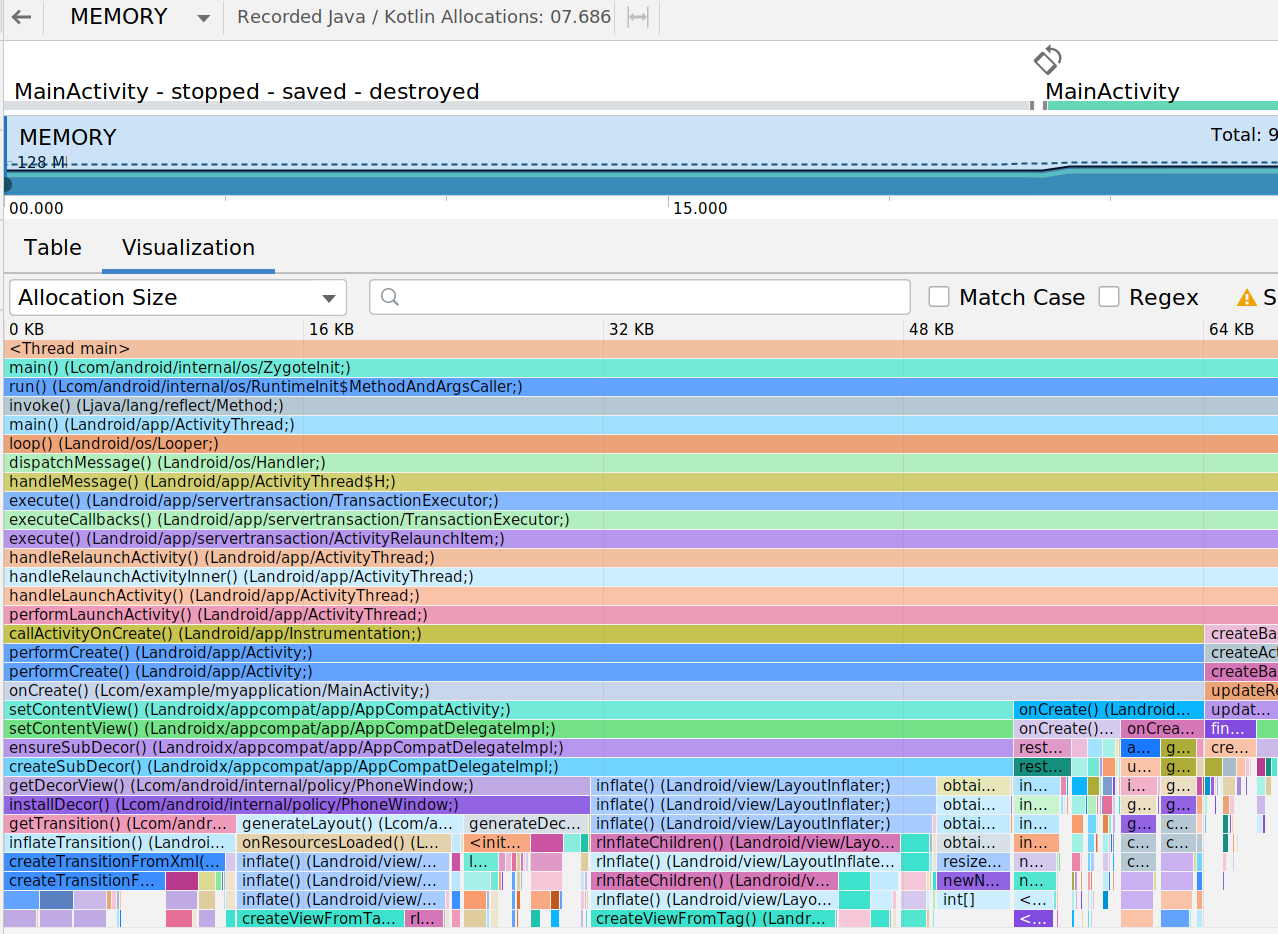
* Java – Memorija koju alociraju objekti iz Java/Kotlin koda
* Native – Memorija koju alociraju objekti iz C/C++ koda
* Graphics – Memorija koriscena za baferovanje grafickih objekata za prikaz piksela na ekranu, ukljucujuci GL surface, GL texture,…
* Stack – Memorija koriscena od strane native I Java stack-ova u aplikaciji. Ova sekcija se povezuje sa tim koliko thread-ova aplikacija izvrsava
* Code – Memorija koju aplikacija koristi za cuvanje koda I potrebnih resursa poput dex bytecode-a, optimizovanog ili kompajliranog dex code-a, .so biblioteka I slicno
* Others – Memorija koriscena od strane aplikacije koju sistem ne moze da kategorizuje
* Allocated – Broj Java/Kotlin objekata alociranih od strane aplikacije. Ovo ne ukljucuje objekte alocirane od strane C ili C++ koda.

# **Pregled memorijskih alokacija**

Memorijske alokacije pokazuju kako je svaki Java objekat I JNI referenca alocirana u memoriji. Konkretno, Memory Profiler nam moze pokazati sledece stvari vezano za alokaciju objekta:

* Koji tipovi objekata su alocirani i koliko prostora zauzimaju
* Stack trace za svaku alokaciju, ukljucujeci i koji thread je zaduzen
* Kada su objekti bili dealocirani

Da bi snimili Java i Kotlin alokacija, potrebno je izabrati opciju **Record Java/Kotlin allocations**, I onda opciju **Record**. Ukoliko uredjaj ima Android 8 ili noviji kao operativni sistem, Memory Profiler UI transaction screen ce se otvoriti kao poseban ekran.



Nakon sto selektujemo regiju timeline-a (ili zavrsimo sesiju snimanja na uredjaju koji ima Android 7.1 ili novije), prikazace nam se lista alociranih objekata, grupisanih po class name-u I sortiranim po velicini heap-a.

Da bi se analizirao taj snimak, potrebno je ispratiti sledece korake:

1. Potrebno je proci kroz listu I pronaci objekte koji imaju neobicno veliku kolicinu heap memorije I koji su potencijalno uzroci curenja memorije. Da bismo lakse pronasli potrebnu klasu, mozemo kliknuti na **Class Name** I da sortiramo po alfabetu. Nakon toga kada kliknemo na class name. **Instance View** panel sa desne strane, prikazujuci svaku instancu te klase.
2. Na **InstanceView** panelu, kliknemo na instancu. **Call Stack** tab ce se prikazati ispod I prikazano je mesto gde je instance alocirana I u kojoj niti.
3. U **Call Stack** tabu, desni klik I opcija **Jump to Source** ce otvoriti kod u editoru.

Graphical user interface, website

Description automatically generated

Ova dva menija se mogu koristiti za pregled liste alociranih objekata I tada se moze odabrati koji heap treba istraziti I kako organizovati podatke.

Iz menija mozemo odabrati koji heap treba istraziti:

* **Default heap –** Kada heap nije specificiran sistemom.
* **Image heap –** System boot image, koji sadrzi klase koje se ucitavaju tokom boot time-a. Alokacije cuvane ovde garantovane se ne pomeraju
* **Zygote heap –** copy on write heap gde je aplikacioni process izdvojen iz Android sistema.
* **App heap –** Primarni heap gde aplikacija alocira memoriju
* **JNI heap –** Heap koji pokazuje gde Java Native Interface (JNI) smesta svoje reference

Da bi se unapredio perfomans aplikacije prilikom profilisanja, memory profiler periodicno uzorkuje (skladisti) alokacije memorije. Kada testiramo aplikaciju na uredjaju koji koriste API 26 ili noviji, ovo ponasanje je mogu prilagoditi uz pomoc opcije Allocation Tracking. Moguce su sledece vrednosti:

* **Full –** Skladisti sve alokacije u memoriju. To je default ponasanje u Android Studio 3.2 ili novijim. Ukoliko imamo aplikaciju koja alocira puno objekata, bice primetno promenjeno ponasanje aplikacije prilikom profajlinga u vidu usporenog rada.
* **Sampled –** Skladisti alokacije objekata u memoriji u regularnim intervalima. Ovo je default opcija I ima manji uticaj na perfomanse aplikacije prilikom profilisanja.
* **Off –** Prekida pracenje memorijskih alokacija.

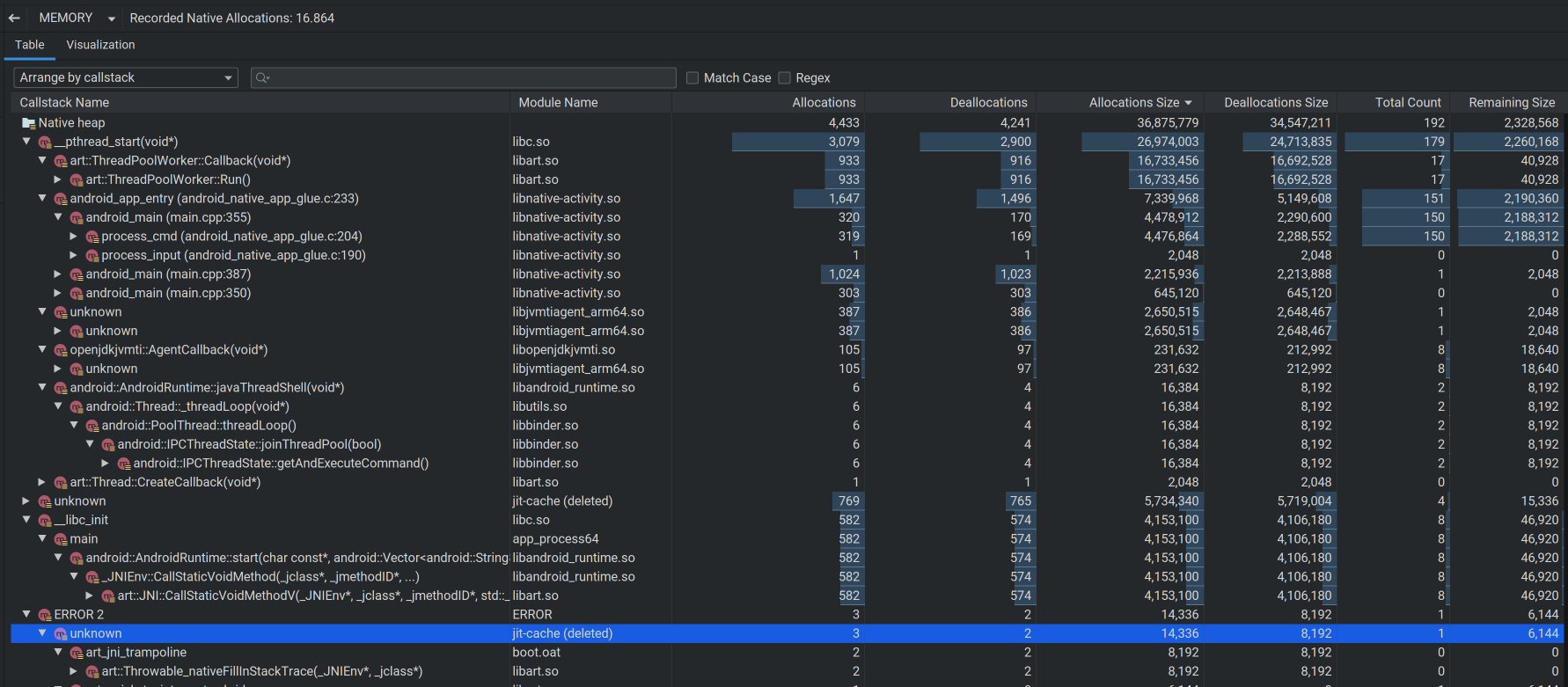
### Pregled globalnih JNI reference

*Java Native Interface (JNI)* je framework koji omogucava komunikaciju izmedju Java koda I native (c/c++) koda. JNI Referencama se upravlja rucno iz native code-a, pa je moguce da Java objekti koji su korisceni od strane native koda ostanu predugo alocirani. Neki objekti na Java heapu mogu postati nedostupni ako je JNI referenca obrisana bez prethodnog njihovog brisanja. Takodje, moguce je izazvati prekoracenje limita za cuvanje JNI reference.

Da bi se ovaj problem lakse nasao I resio, mozemo koristiti **JNI heap** u Memory Profiler-u za pregled svih globalnih JNI reference I filterisanje po Java tipovima I native call stack-ovima. Sa ovom informacijom, moze pronaci kada I gde se globalne JNI reference kreiraju I brisu.

Dok se aplikacija izvrsava, potrebno je odabrati deo timeline-a koji zelimo da proverimo I onda biramo JNI heap iz liste. Tada se ovi objekti mogu istraziti kao I obicni java objekti I analizirati uz pomoc Allocation Call Stack-a.

### Native Memory Profiler



Android Studio Memory Profiler ukljucuje I Native Memory Profiler za aplikacije smestene na fizickom uredjaju koji pokrece Android 10. Native Memory Profiler prati alokacije/delokacije objekata u native code-u za specifican vremenski period I pruza sledece informacije:

* **Allocations** – Broj objekata alociranih sa metodom malloc() ili sa new operatorom tokom specificiranog vremenskog perioda.
* **Deallocations** – Broj objekata delociran sa metodama free() ili delete operatorom tokom specificiranog vremenskog perioda.
* **Allocations size** – Sumirana velicina u bajtovima za sve alokacije u odabranom vremenskom period
* **Deallocations size** – Sumirana velicina u bajtovima oslobodjene memorije tokom odredjenog vremenskog perioda.
* **Total count** – vrednost u Allocations koloni minus vrednost iz Deallocations kolone
* **Remaining size** – Allocations Size – Deallocations Size

Da bi se snimile native alokacije na uredjajima koji pokrecu Android 10 ili noviji, treba odabrati opciju **Record Native Allocations**, I tada selektovati **Record**. Snimanje ce trajati sve do momenta kada kliknemo na Stop, nakon cega ce se pokazati odvojeni ekran sa podacima I tranzicijama.

# **Snimanje heap dump-a**

Heap dump prikazuje koji objekti u memoriji koriste memoriju u momentu kada snimamo heap dump. Posebno nakon produzene korisnicke sesije, heap dump moze pomoci u identifikaciji memory leak-ova tako sto prikazuje objekte koji su jos u memoriji a ne bi trebalo da se nalaze tamo.

Kada snimamo heap dump, mozemo dobiti sledece informacije:

* Koji tipovi objekta aplikacije su alocirani, I kako pojedinacno objekata
* Koliko memorije svaki objekat koristi
* Gde se nalaze reference svakog objekta koje se koriste u kodu
* Call stack ka objektu koji je alociran.

Da bismo zapoceli snimanje, potrebno je kliknuti **Capture heap dump**, I onda selektovati **Record.** Tokom sladistenja heap-a, kolicina java memorije moze privremeno porasti. Ovo je normalno posto se snimanje desava u istom procesu kao I sama aplikacija. Kada profajler zavrsi snimanje heap dump-a, Memory Profiler UI ce otvoriti ekran sa snimljenim podacima:

Graphical user interface, text, application, email

Description automatically generated

Ukoliko nam trebaju preciznije informacije, mozemo kreirati heap dump u kriticnom trenutku rada aplikacije time sto cemo pozvati metodu **dumpHprofData()** u kodu.

U listi klasa, mozemo videte sledece podatke:

* **Allocations –** Broj alokacija u heapu
* **Native size –** Ukupna kolicina native memorije koriscenja od strane datog tipa objekta. Ova kolona je dostupna samo za Android 7 ili novije
* **Shallow size –** Totalna kolicina Java memorije koriscena od strane ovog tipa objekta
* **Retained size –** Totalna kolicina memorije koja se zadrzava zbog svih instance ove klase.

Kada kliknemo na class name, otvara se **Instance view** prozor gde ce biti prikazane sve instance sa sledecim informacijama:

* **Depth –** najkraci broj skokova od bilo kog Garbage Collectora do izabrane instance
* **Native size –** Velicina instance native memoriji.
* **Shallow size –** Velicina instance u Java memoriji
* **Retained size –** Velicina memorije gde ova instance dominira.

Graphical user interface, application

Description automatically generated

Da bismo istrazili heap, treba ispratiti sledece korake:

1. Proci kroz listu u cilju pronalaska objekta sa neobicno velikom kolicinom heap memorije koji potencijalno mogu procureti. Da bi se pronasla poznata klasa, klik na **Class name** header ce sortirati klase po alfabetu. Tada mozemo kliknut na class name I otvorice se **InstanceView** u kome su prikazane sve instance date klase.
2. Kada odaberemo instancu u **InstanceView**, **Reference tab** ce se prikazati sa svim referencama ka tom objektu.
3. U **Reference tabu**, ukoliko zakljucimo da odredjena referenca moze dovesti do leak-a, desni klik I opcija **Go to Instance** ce dovesti do odgovarajuce instance u heap dump-u prikazivajuci podatke o toj instanci.

U heap dumpu trebamo traziti za memory leak-ove koji mogu biti posledice sledecih situacija:

* Long lived reference za Activity, Context, View, Drawable ili druge objekte koji mogu sadrzati reference ka Activity ili Context container-u.
* Non static inner klase, poput Runnable koje takodje mogu sadrzati Activity instance
* Cache koji drzi objekte duze nego sto je potrebno

## Cuvanje heap dump-a u HPROF fajlu

Nakon sto snimimo heap dump, podaci su dostupni u Memory Profiler samo dok je profiler pokrenut. Kada zavrsimo profiling sesiju, heap dump se brise. Ukoliko zelimo da sacuvamo podatke da bismo kasnije nastavili sa pregledom, mozemo ga sacuvati kao HPROF fajl. U Android Studiju 3.1 ili stariji, **Export capture to file** se nalazi na levoj strain toolbara, dok je na novijim verzijama Android Studija na desnoj strain toolbara dugme **Export Heap Dump** dugme.

# **Detekcija curenja memorije u Memory Profiler-u**

Kada analiziramo heap dump u Memory Profiler-u, Android Studio moze prikazivati podatke za koje smatra da mogu dovesti do curenja memorije, a vezani su za Activity I Fragment objekte u aplikaciji. Tipovi ovih podataka koje filter prikazuje su sledeci:

* **Activity** instance koje su trebale biti unistene ali su I dalje referencirane
* **Fragment** instance koje nemaju validan FragmentManager ali su I dalje referencirane

U odredjenim situacijama, kao sto su sledece, filter moze dati lazne pozitivne rezultate:

* **Fragment** je kreiran, ali jos uvek nije koriscen
* **Fragment** je bio kesiran ali nije bio deo **FragmentTransaction**

