

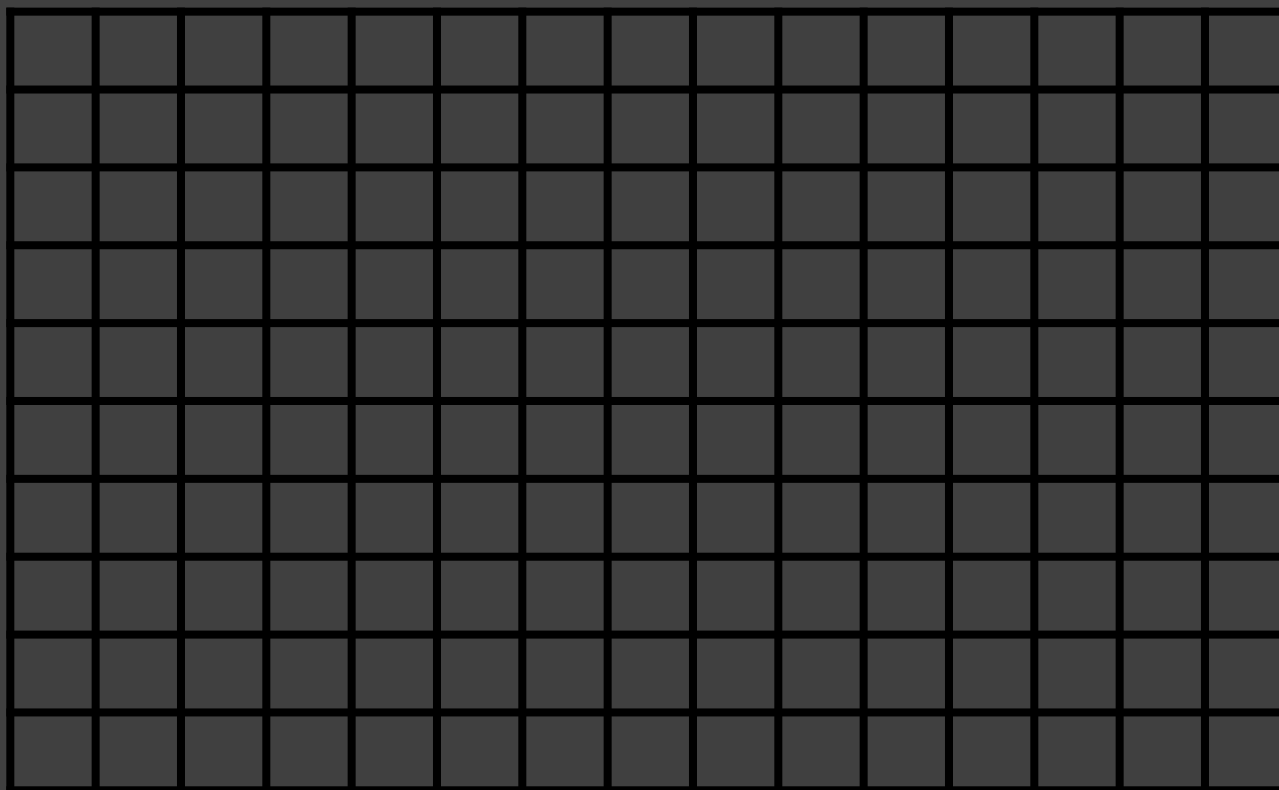


WSTĘP DO PROGRAMOWANIA URZĄDZEŃ MOBILNYCH KOTLIN, JAVA

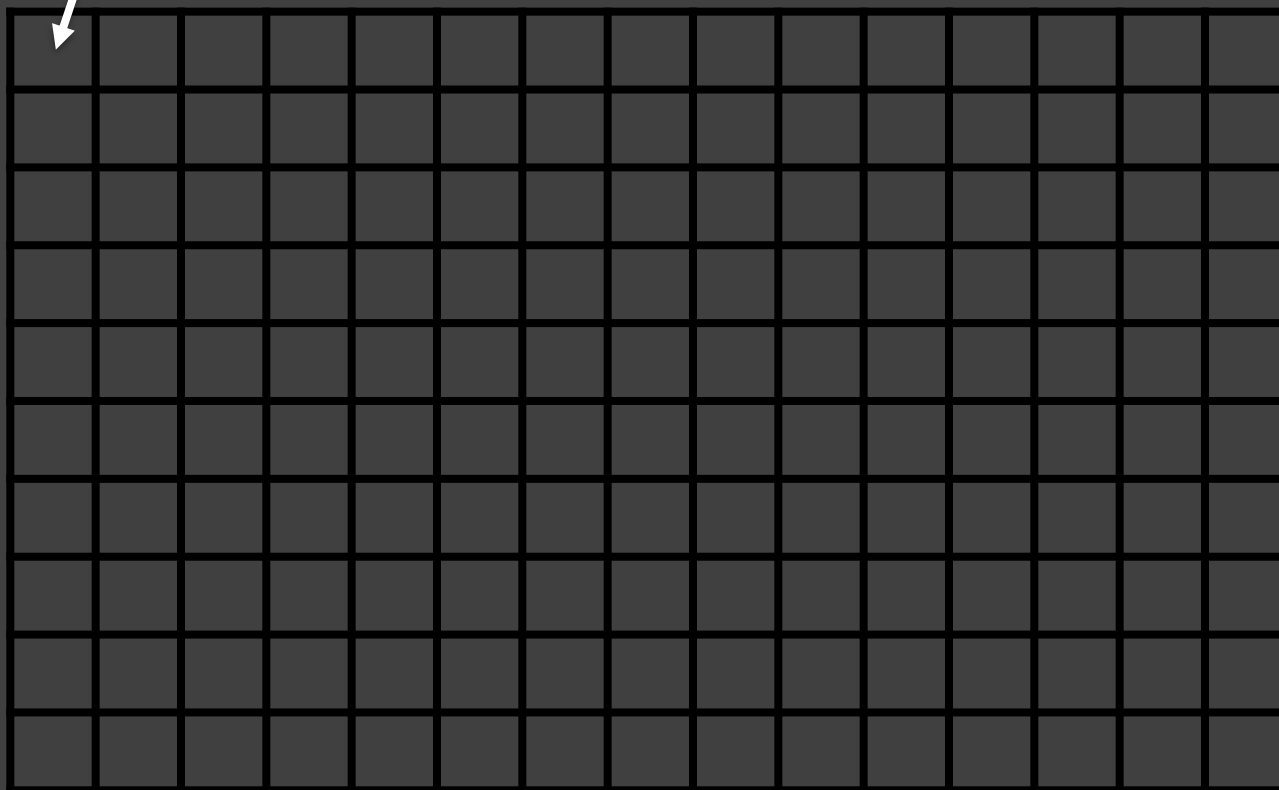
WYKŁAD 2

- Garbage Collection

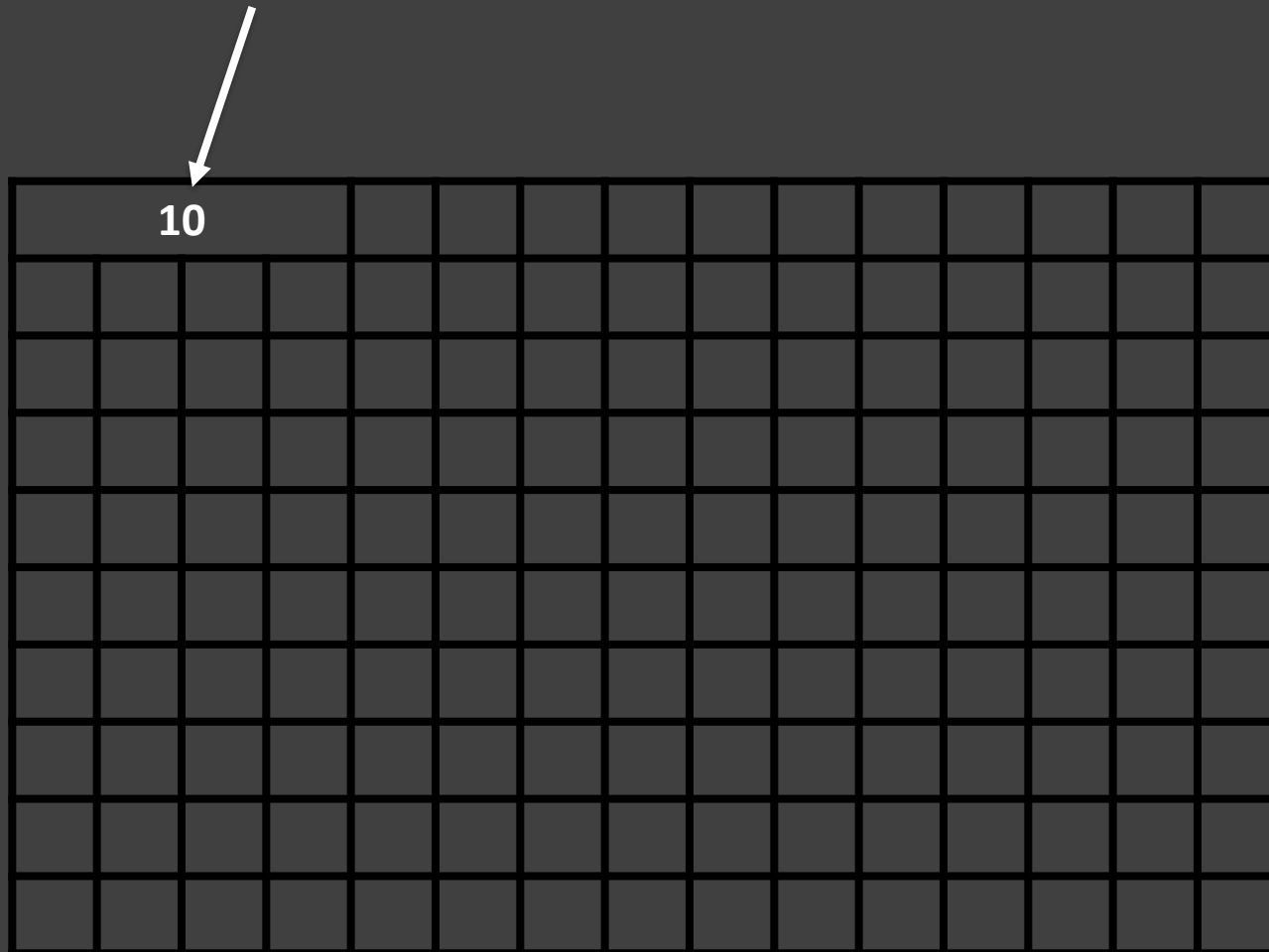
- Bool - 1 byte
- Char - 1 byte
- Double - 8 bytes
- Float - 4 bytes
- Int - 4 bytes
- Long - 8 bytes



Bajt = 8 bit

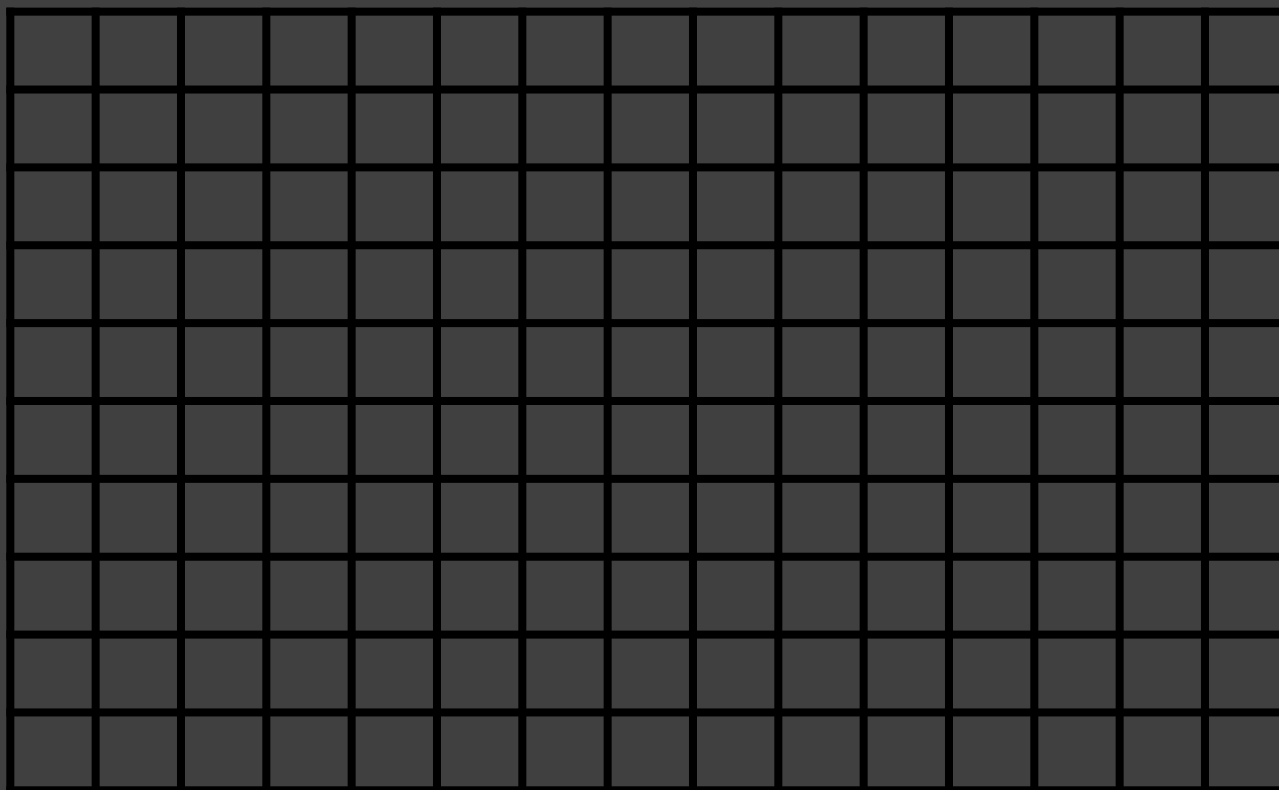


int = 4 bajt



[illegible]

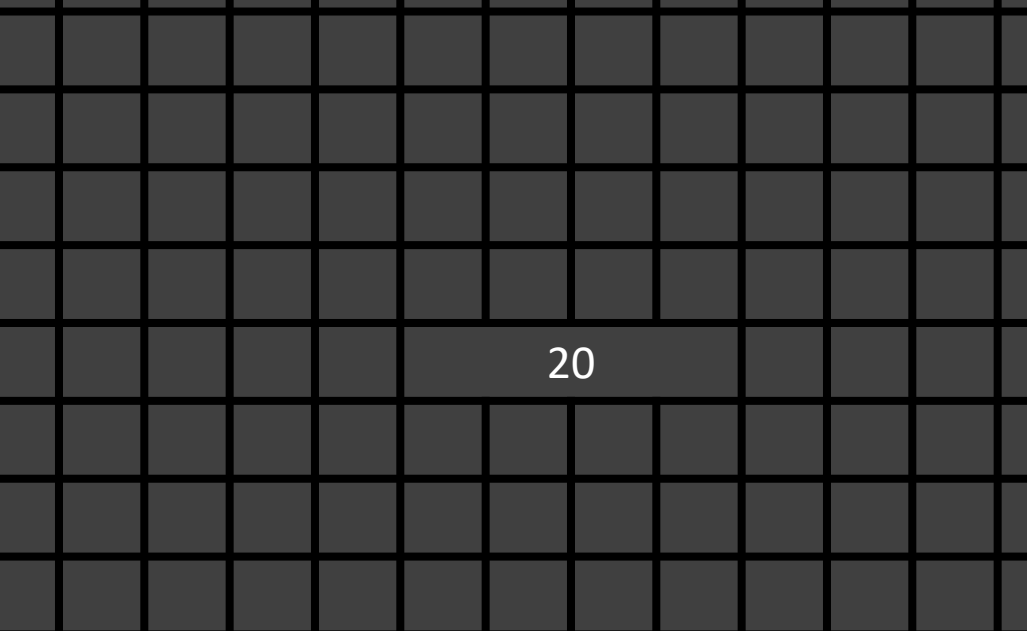
int score = 20



20

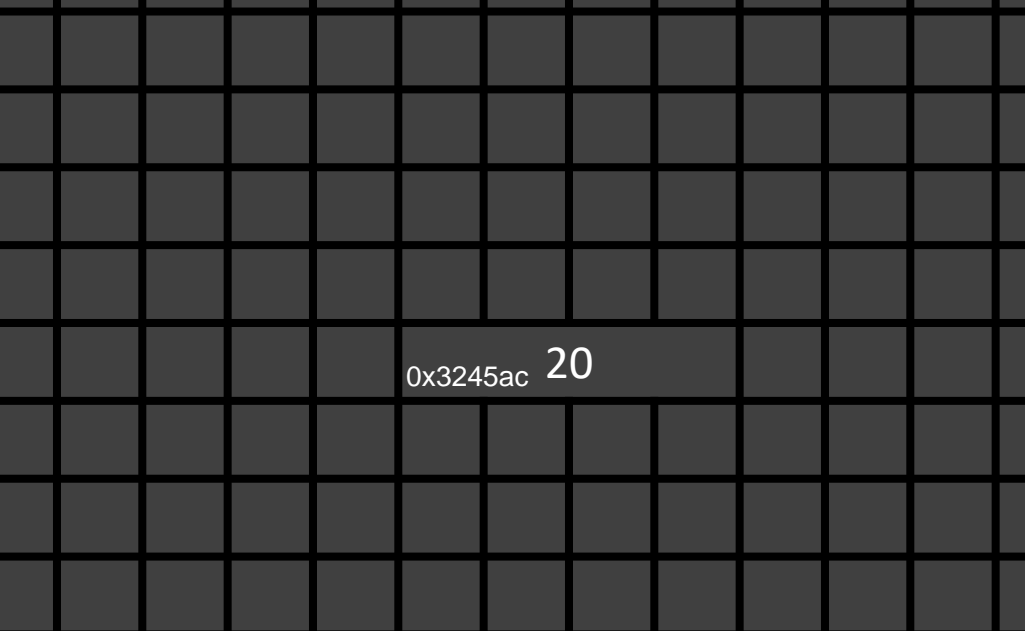
& - pokaż gdzie zmienna jest umieszczona

20



20

0x3245ac

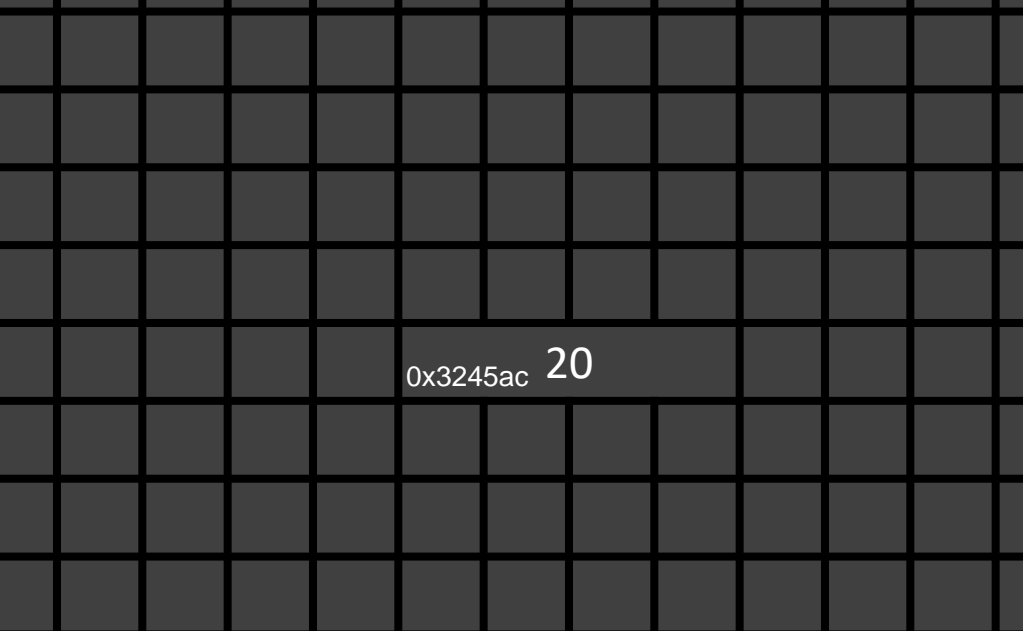


0x3245ac 20

*** - pokaż co jest pod adresem**

[illegible]

20

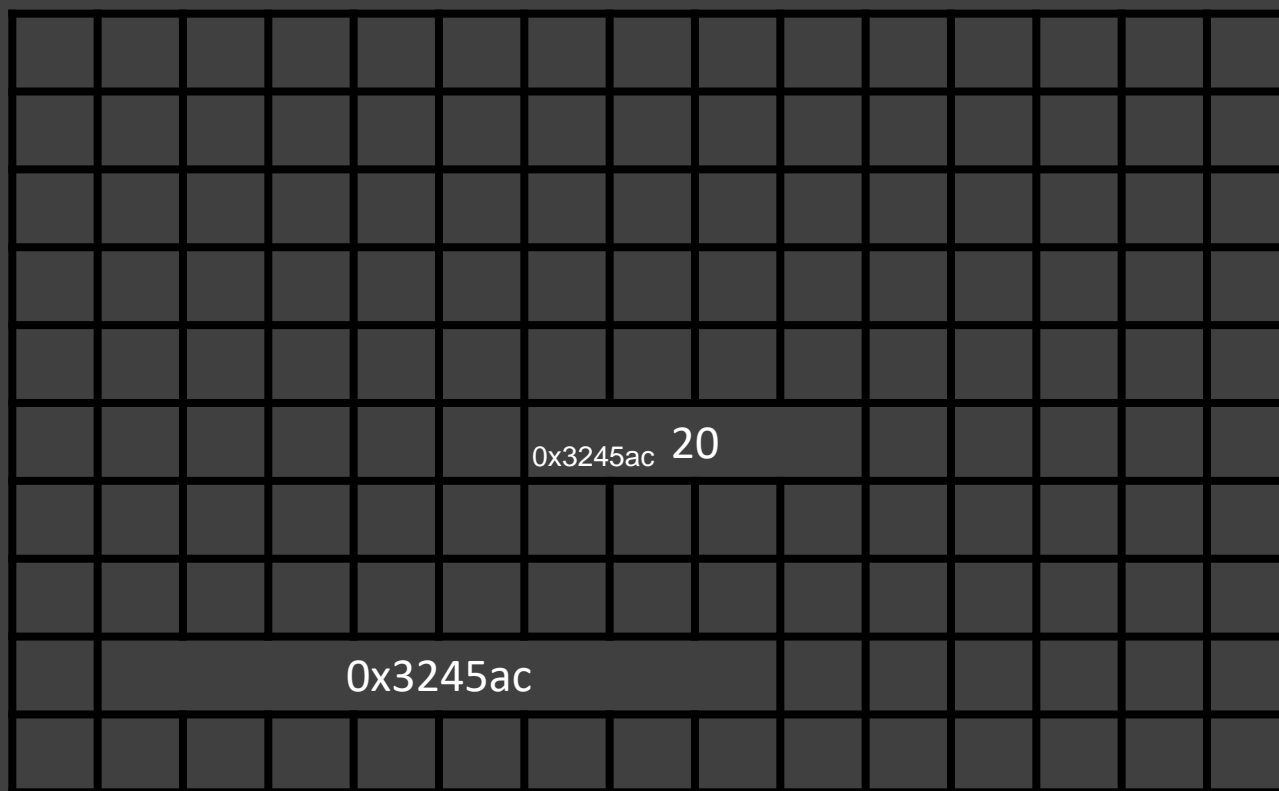


0x3245ac 20

- | | | | |
|----------|-----------|-----------|-----------|
| • Bool | - 1 byte | • Bool* | - 8 bytes |
| • Char | - 1 byte | • Char* | - 8 bytes |
| • Double | - 8 bytes | • Double* | - 8 bytes |
| • Float | - 4 bytes | • Float* | - 8 bytes |
| • Int | - 4 bytes | • Int* | - 8 bytes |
| • Long | - 8 bytes | • Long* | - 8 bytes |
| | | • Void* | - 8 bytes |

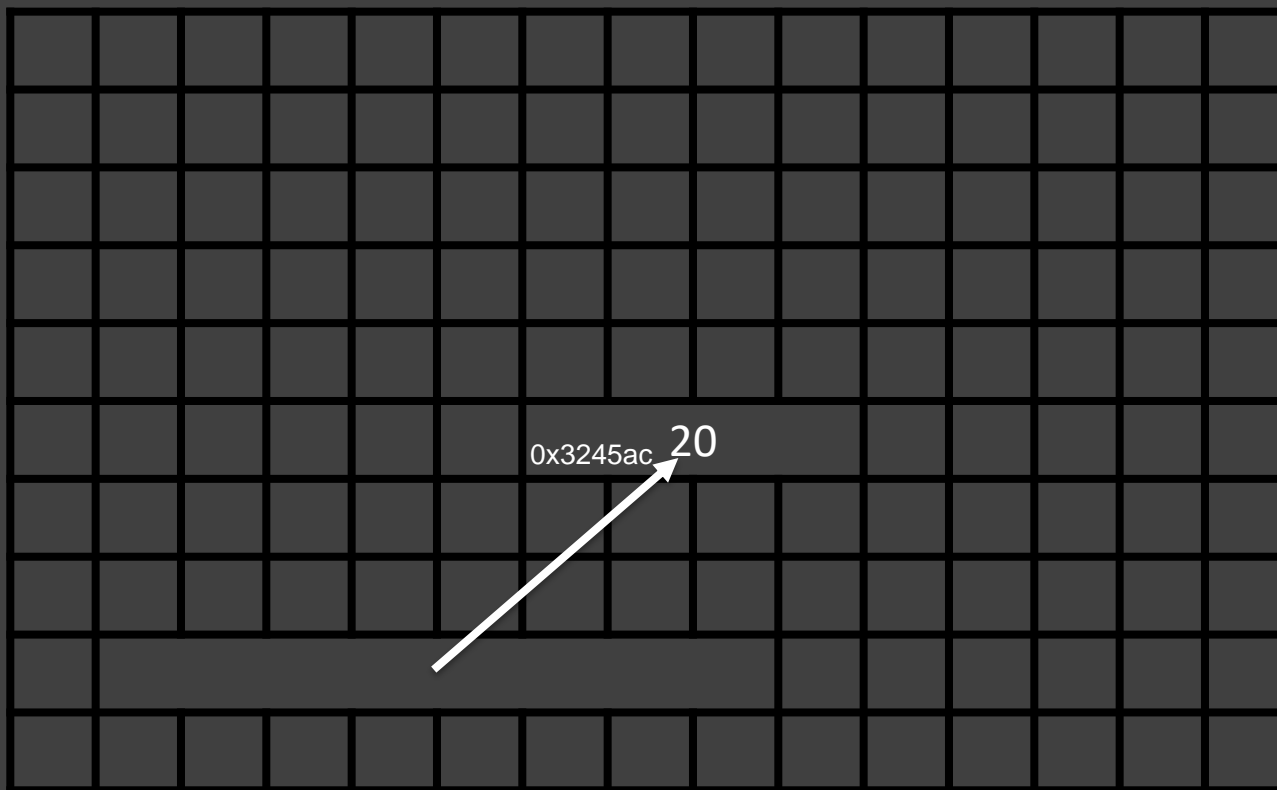
[illegible]

```
int n = 20;  
int *p = &n;  
printf („%i \n”, *p);  
20
```





```
int n = 20;  
int *p = &n;  
printf („%i \n”, *p);  
20
```





```
int n [2] = {2, 3};  
Int *p = malloc(2 * sizeof(int));  
  
if(p == NULL) {  
    exit(0);  
}  
  
for(int i = 0; i < n.length; ++i) {  
    p [ i ] = n [ i ]  
}  
  
free(p)
```

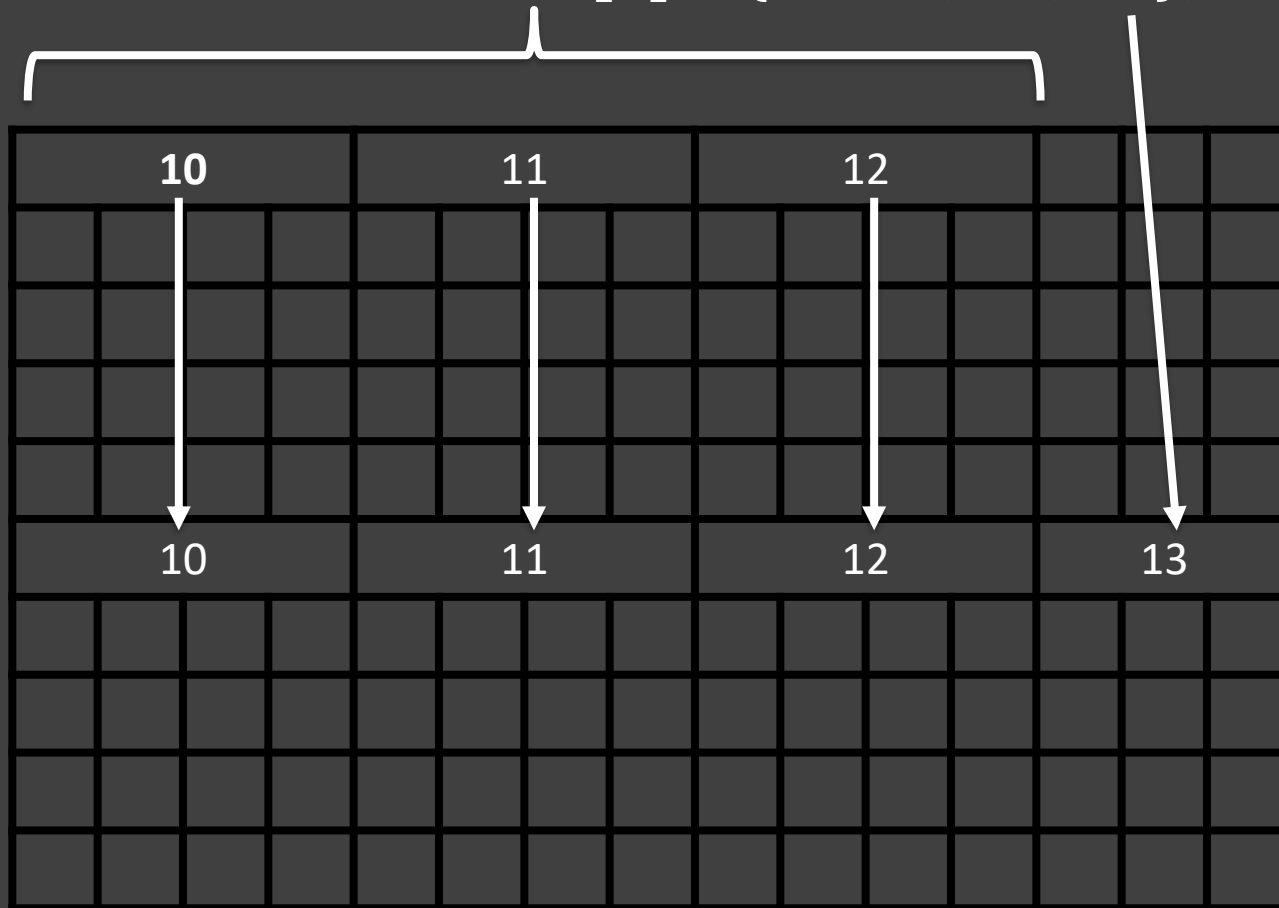
[illegible]

[illegible]

A 10x15 grid with a bracket above the first three columns labeled 10, 11, and 12. The number 13 is in the sixth column of the sixth row.

ArrayList

```
int scores [3] = {10, 11, 12};  
int cscores [4] = {10, 11, 12, 13};
```



[illegible]



LinkedList

```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

number							
next							


```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

1							
null							
0x123							



LinkedList

```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

```
node *n = malloc(sizeof(node));
```

1							
null							
0x123							



LinkedList

```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

```
node *n = malloc(sizeof(node));  
if (n != NULL){  
    (*n).number = 1  
}
```

1							
null							
0x123							



LinkedList

```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

```
node *n = malloc(sizeof(node));  
if (n != NULL){  
    (*n).number = 1  
    n -> next = NULL  
}
```

1							
null							
0x123							

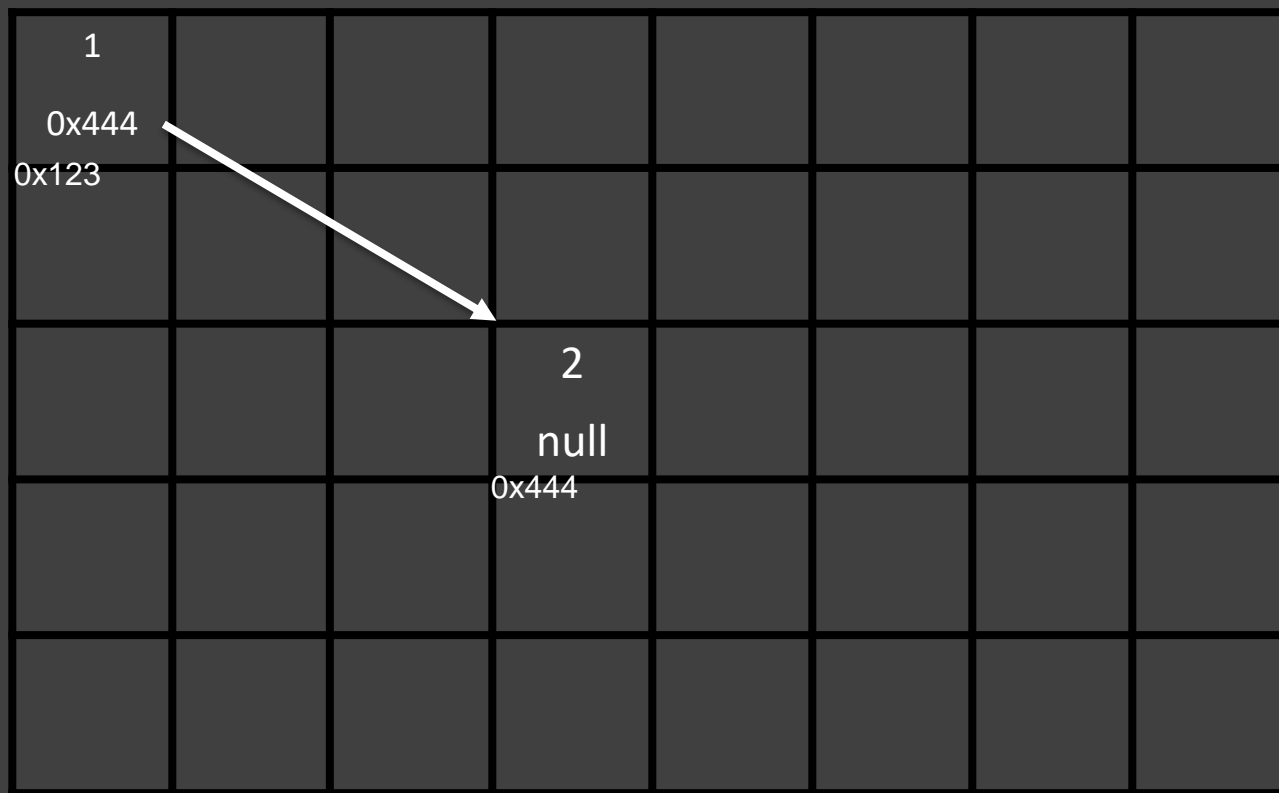


LinkedList

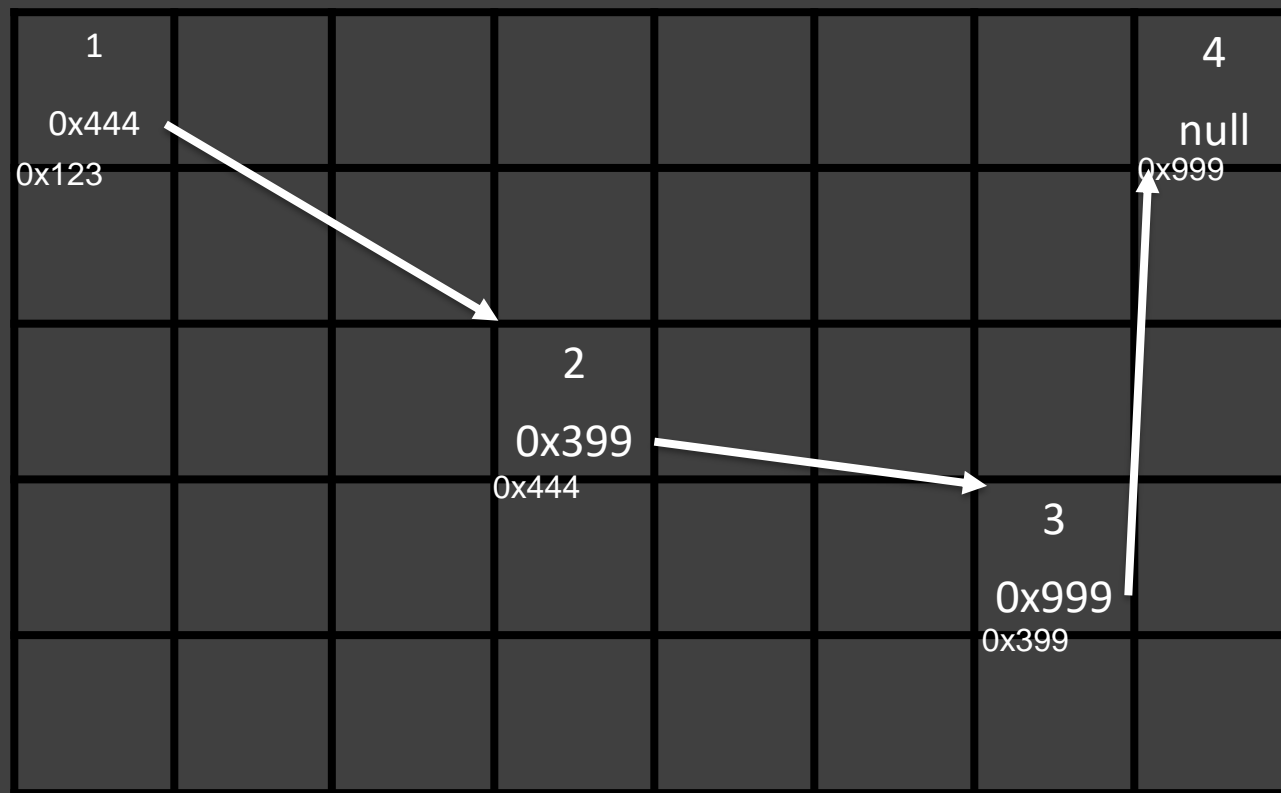
```
typedef struct node {  
    int number;  
    struct node *next;  
}  
node;
```

```
node *n = malloc(sizeof(node));  
if (n != NULL){  
    n -> number = 1  
    n -> next = NULL  
}
```

1							
null							
0x123							



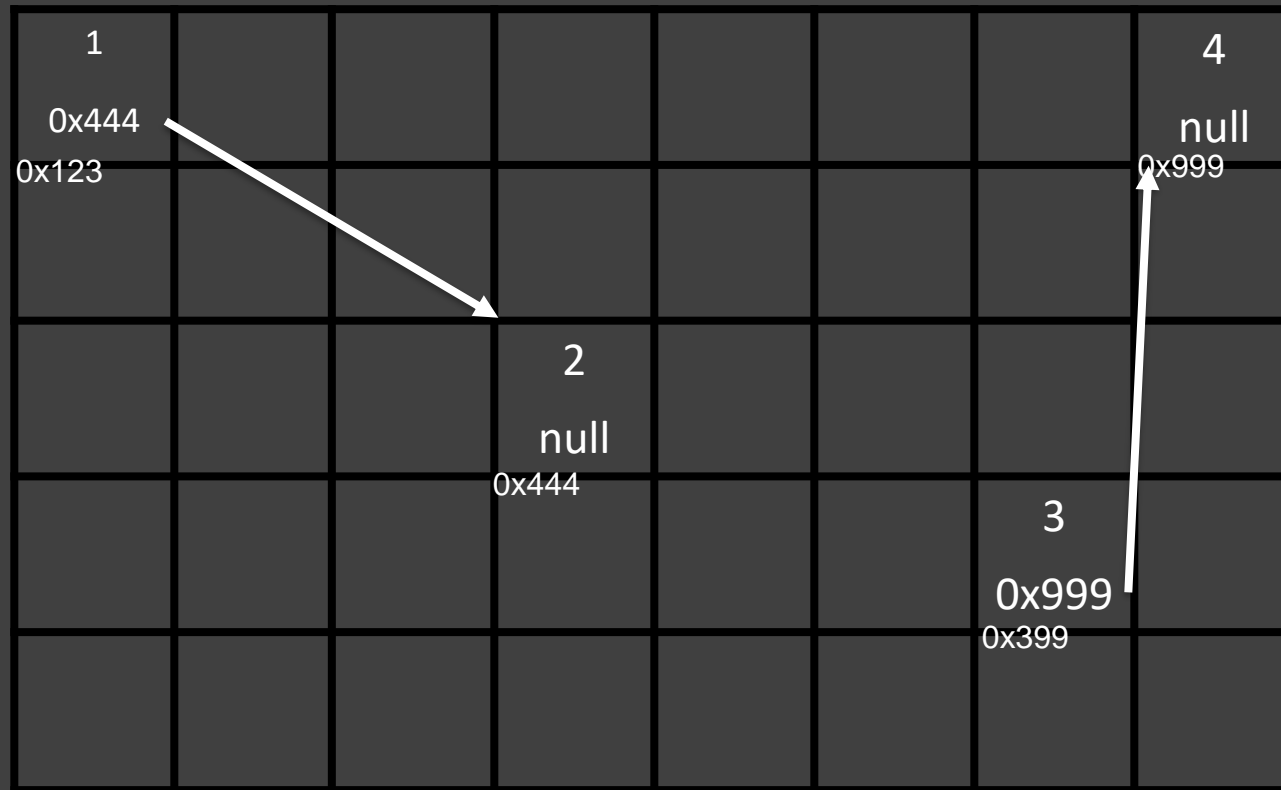
LinkedList





Memory Leak

free(n)



W językach takich jak C/C++ musimy zarządzać pamięcią - zaalokować oraz zwolnić.

Malloc()

Realloc()

Calloc()

free(n)

destructors

Java wprowadziła automatyczne zarządzanie pamięcią – Garbage Collector.

Usuwa obiekty które już nie są używane.

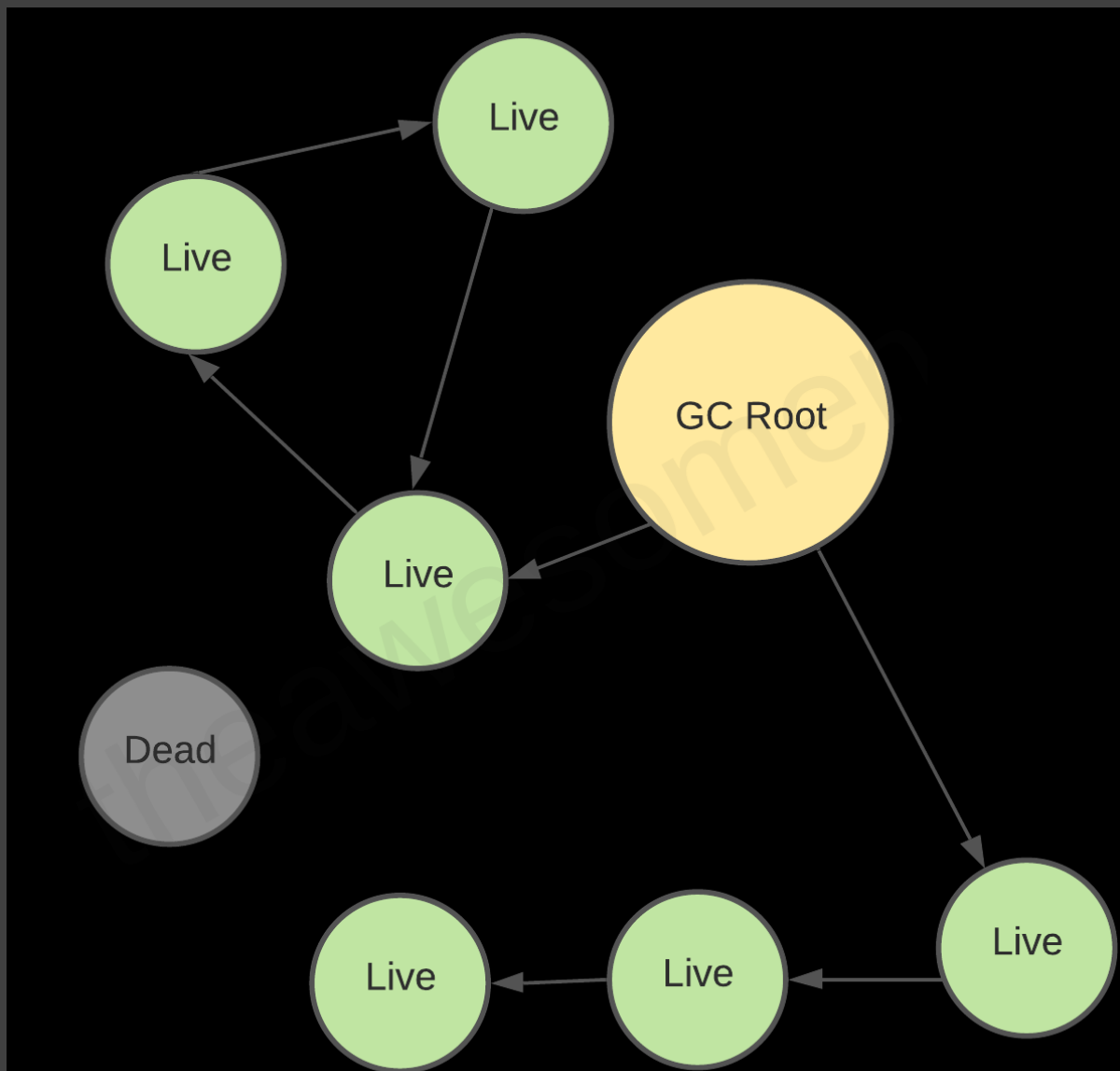
- **Live objects – obiekty osiągalne (do którego odwołuje się inny obiekt)**
- **Dead objects – obiekty nieosiągalne (do którego nie odwołuje się żaden inny obiekt)**

Zbieranie nieużytków jest realizowane przez demon (wątek działający niezależnie od użytkownika) – Garbage Collector

Obiekty są alokowane (słowo kluczowe new) na stosie.

- **Java 7 – PermGen - Składowe są przechowywane w specjalnie przydzielonej części pamięci odizolowanej od głównej pamięci przeznaczonej na stos**
- **Java 8 – Metaspace – zmiana mająca na celu ograniczenie występowania *OutOfMemory error*. – tutaj pamięć jest przydzielana dynamicznie**

Garbage Collection



Garbage Collector wykorzystuje specjalne obiekty – GC Root. Są to punkty startowe dla procesu zbierania nieużytków.

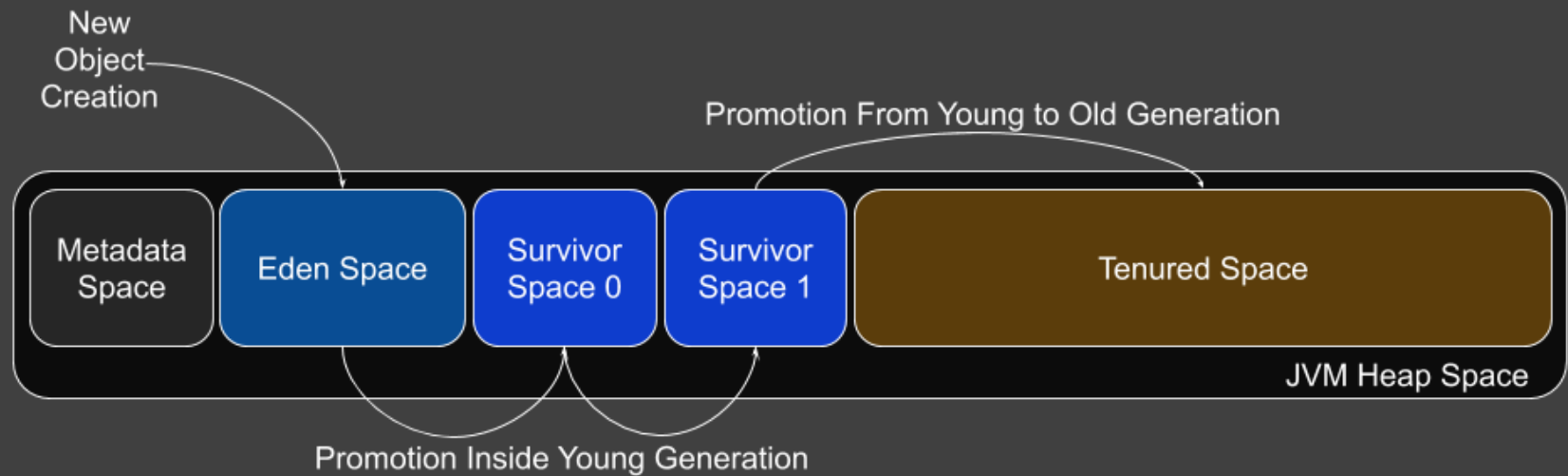
Rodzaje GC Root:

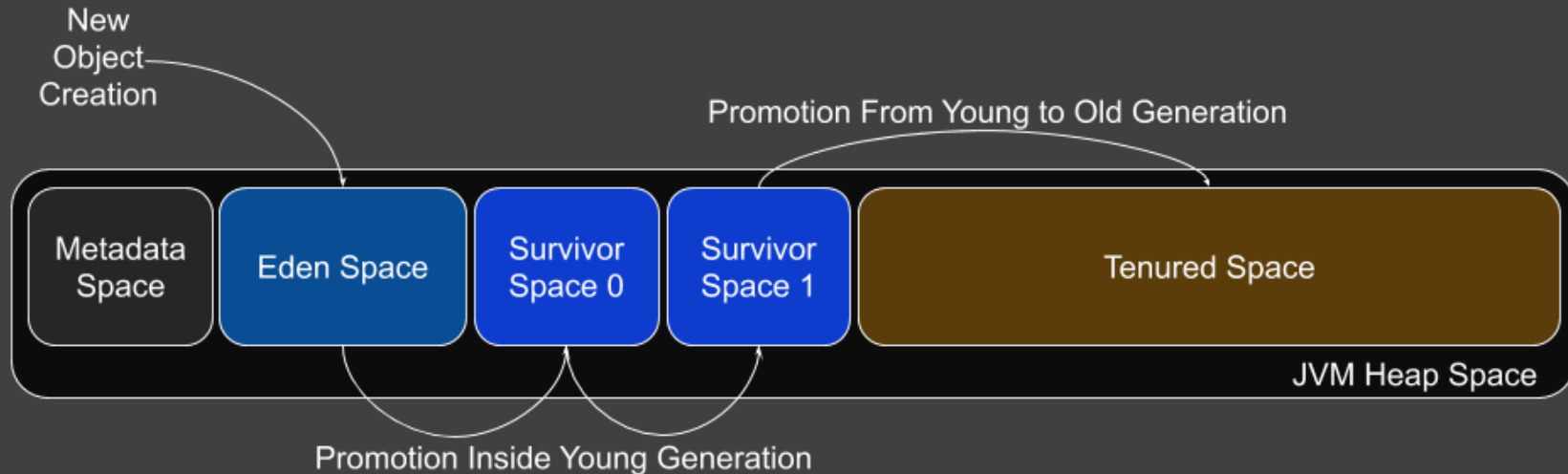
- **Klasa**
- **Lokalny stos – zmienne i parametry**
- **Wątek**
- **JNI**

Kroki gc:

1. **Mark** – gc przechodzi przez graf obiektów rozpoczynając od GC Root – oznacza obiekty jako „żywe” – obiekty bez referencji nie są oznaczane
2. **Sweep** – usuwanie obiektów do których nie można „dojść” przechodząc przez graf
3. **Compacting** – realokacja pamięci

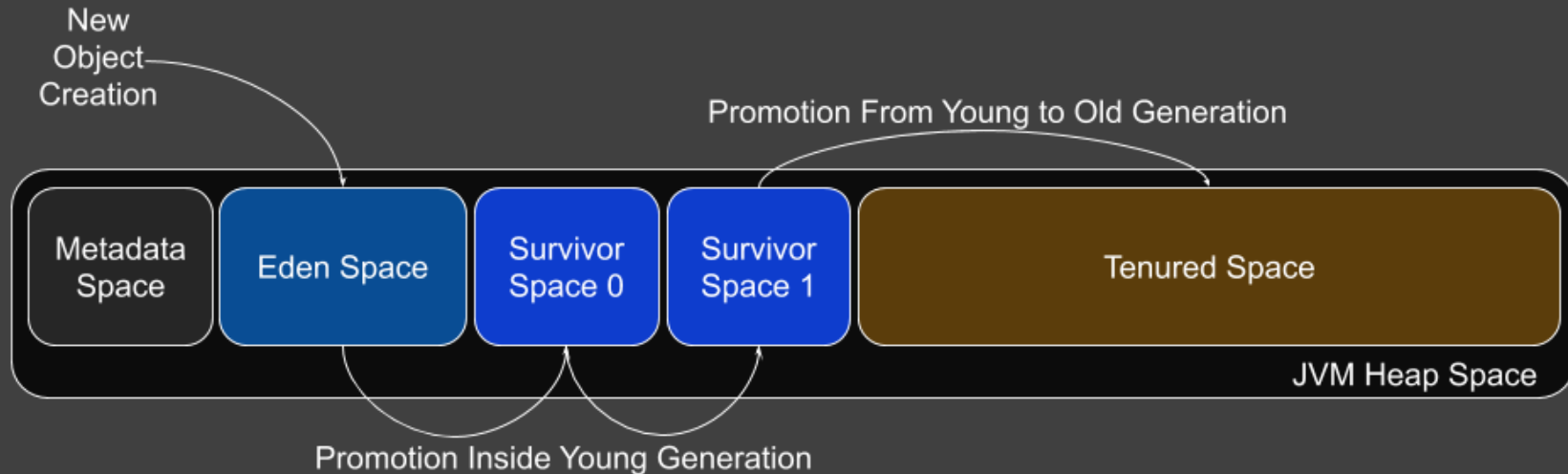
Generational Collectors





Zasada działania:

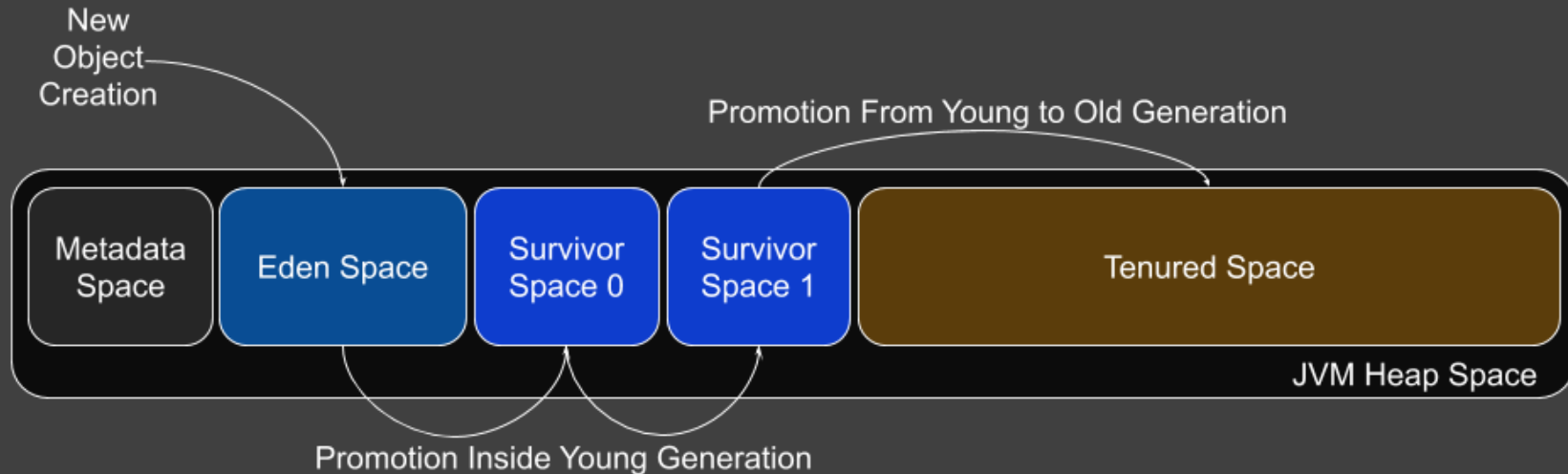
- **Young Generation** – nowe obiekty dopiero utworzone – dzieli się na trzy strefy
 - **Eden**
 - **Survivor space 1**
 - **Survivor space 2**



Zasada działania:

- Wraz z zapełnianiem dostępnej pamięci edenu gc rozpoczyna działanie i wykonuje marking
- Przenosi żywe obiekty do z edenu do s1
- Usuwa nieosiągalne

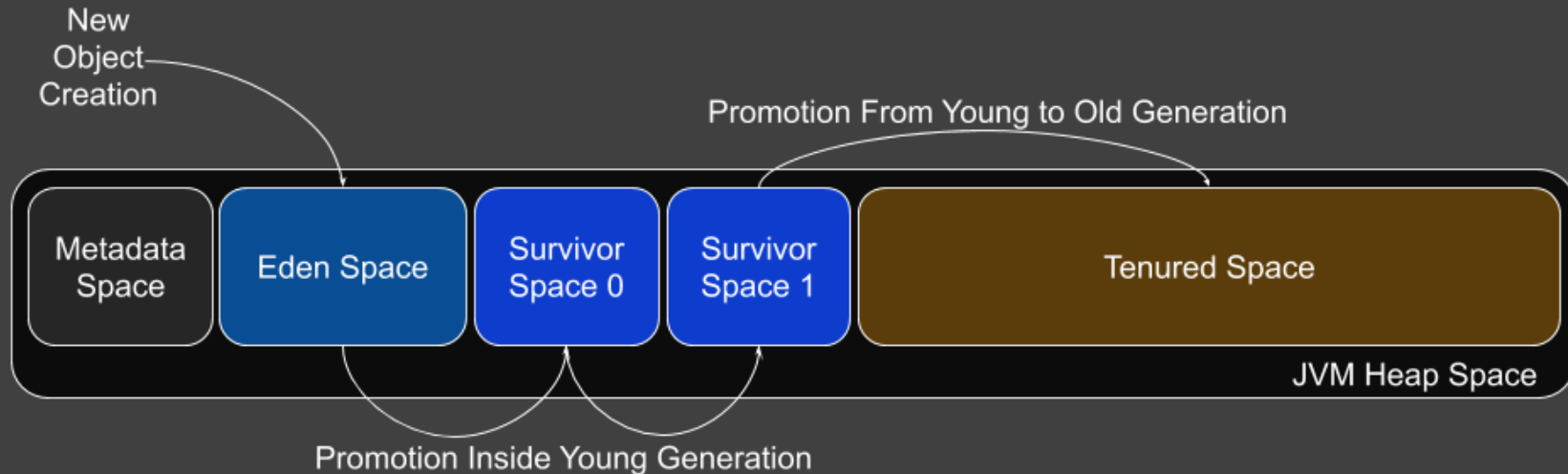
Generational Collectors



Zasada działania:

- W kolejnej iteracji gc wykonuje marking s1 i żywe obiekty przenosi do s2, następnie marking edenu i żywe obiekty przenosi do s1
- W każdej kolejnej iteracji role s1 i s2 są odwracane

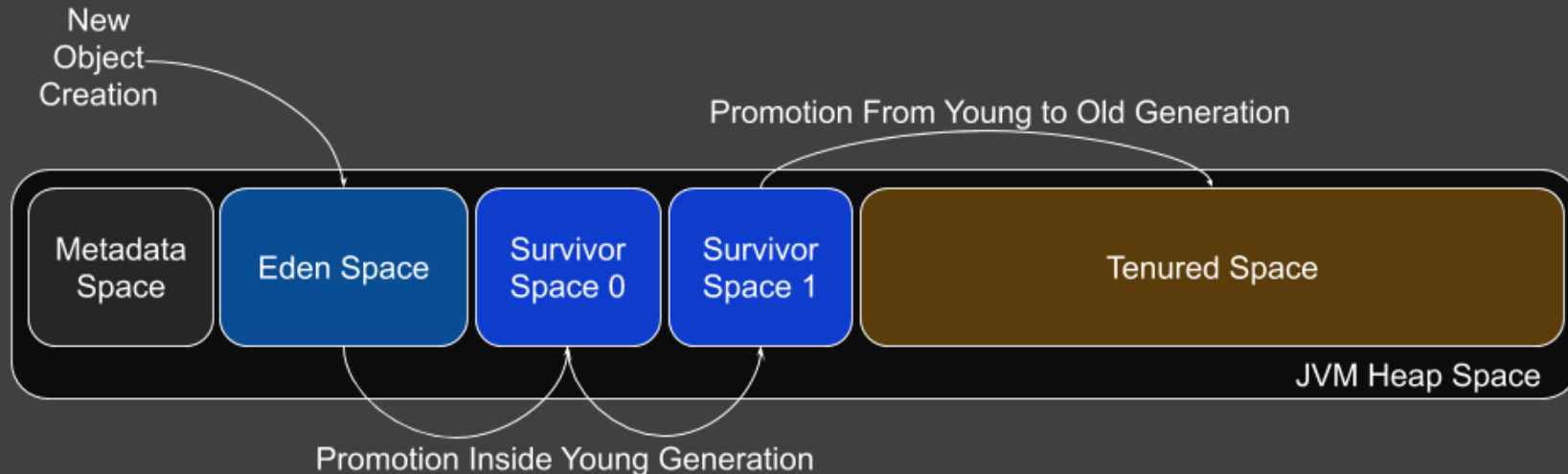
Generational Collectors



Zasada działania:

- **Eden jest czyszczony za każdym razem**
- **Po określonej liczbie iteracji $s1 \rightarrow s2 \rightarrow s1 \dots$ obiekty są promowane do old generation**
- **Iteracje $s1 \rightarrow s2 \rightarrow s1 \dots$ zapobiegają fragmentacji**

Generational Collectors



Zasada działania:

- **Procesy odpowiadające za czyszczenie:**
 - **Młoda generacja – Minor GC**
 - **Stara generacja – Major GC**

1. **Serial Collector – podstawowy gc – jednowątkowy**
2. **Concurrent Collector – wątek działający w trakcie wykonania aplikacji**
3. **Parallel Collector – Wielowątkowy gc**
4. **G1 Garbage Collector – dynamiczne ustalanie młodego regionu przy każdej iteracji – regiony z największą ilością nieużytków zostaną zebrane w pierwszej kolejności**

W kotlinie 1.6.0 pojawił się eksperymentalne podejście do zarządzania pamięcią – może zostać odrzucone.

- **Rozwiązano problem współdzielenia obiektów przez wiele wątków.**
- **Kotlin Multiplatform Mobile (KMM) (alpha) – SDK dla iOS + Android – nowy gc ułatwia wykorzystanie KMM**