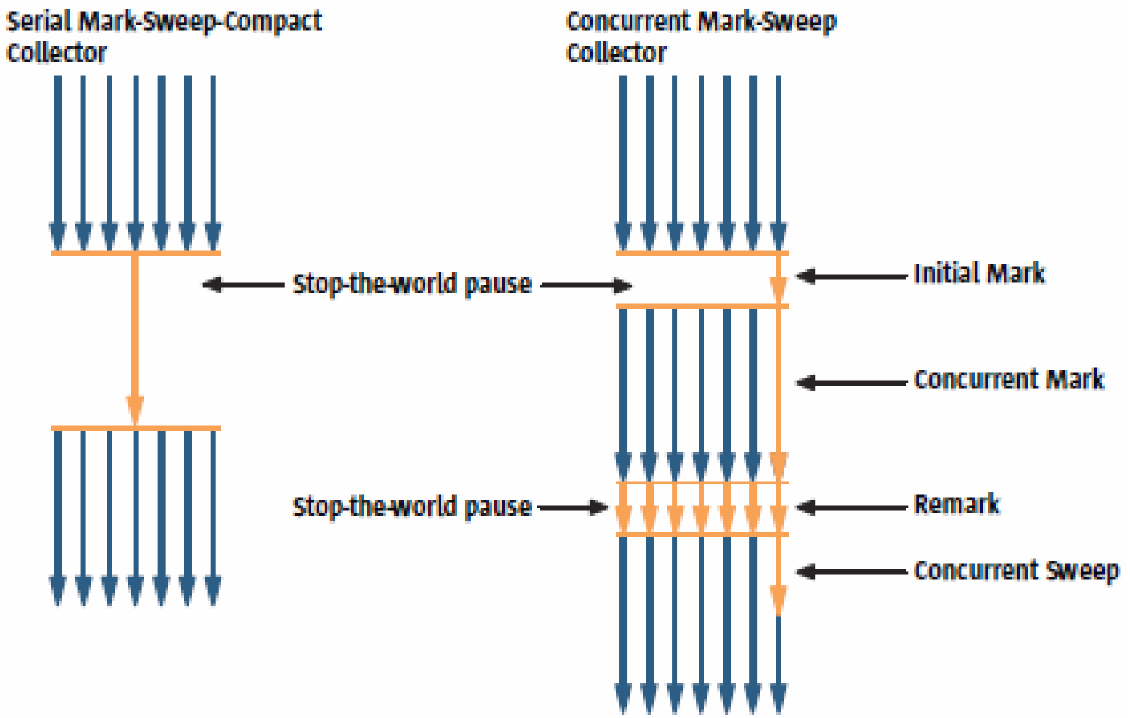
# Architecture of software systems

Course 12: Memory management, garbage collectors

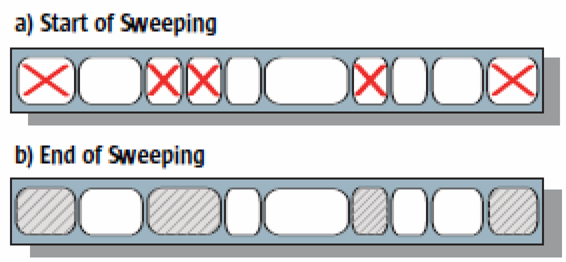
David Šišlák
david.sislak@fel.cvut.cz

# Concurrent mark-sweep (CMS) collector

- low-latency collector for old generation
- reported as **ConcurrentMarkSweep** in memory telemetry
- done concurrently with the application execution
- ***initial mark*** – short pause identifying the initial set of live objects directly reachable from roots; one thread
- ***concurrent mark*** – traversal of objects; all reference modification are monitored by  changed flag
- ***remark*** – revisiting modified objects (overhead); but parallel
- ***concurrent sweep*** – no compaction



# Concurrent mark-sweep (CMS) collector




- » **non-compacting**
- » cannot use bump-the-pointer allocation
- » more **expensive** allocation searching a region

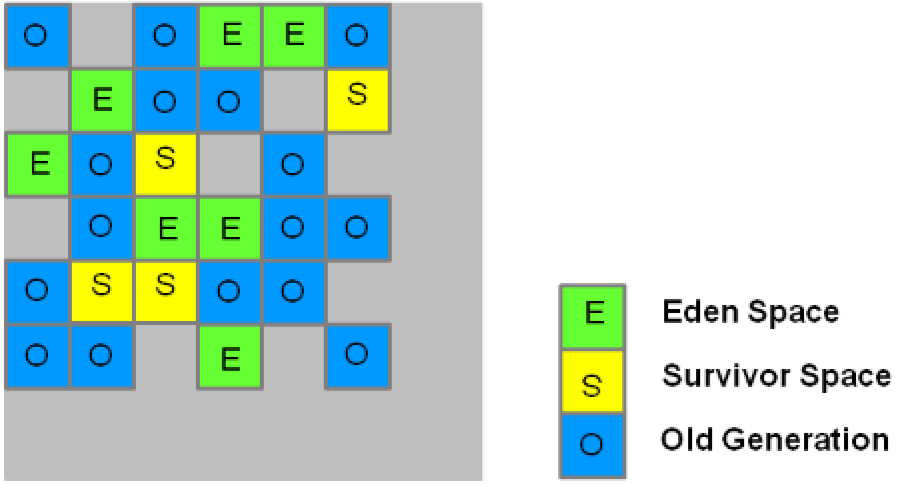
  extra **overhead** to young generation collection doing promotions
- » may split or join free block depending on tracked popular object sizes
- » collector started:
  - adaptively based on previous runs (how long it takes, how many is free)
  - initiating occupancy in percentage

    `-XX:CMSInitiatingOccupancyFraction=n`

    default 68
- » decreases pauses
- » requires **larger heap** due to concurrent collection
- » *incremental mode* – concurrent phases divided into small chunks between young generation collection
- » `-XX:+UseConcMarkSweepGC , -XX:+CMSIncrementalMode`

# mixed G1 collector

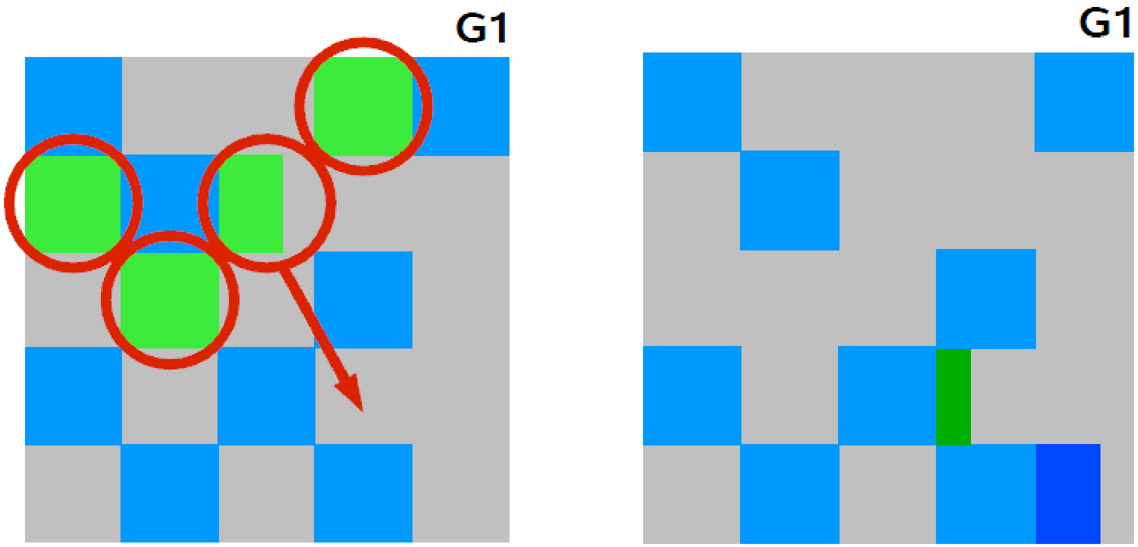
- the latest GC  (introduced in Java 6 update 14)
- whole heap divided into regions (by def. about 2000 regions 1-32MB)
- no explicit separation between generations, only regions are mapped to generational spaces (generation is set of regions, changing in time)




- compacting -> enables bump-the-pointer, TLABs, uses CAS
- compaction = copy live from a region to an empty region
- keep **Humongous regions** (sequence) for objects >=50% regions size
- maintain list of free regions for constant time

# mixed G1 collector – minor GC

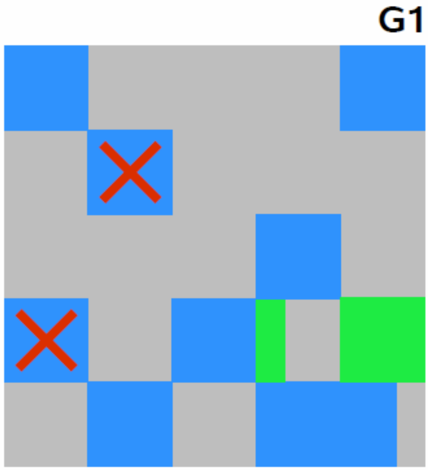
- stop-the-world approach with parallel threads
- live object are copied (from eden and survivor regions) into one or more new survivor regions
- if aging threshold is met => promoted into old generation regions

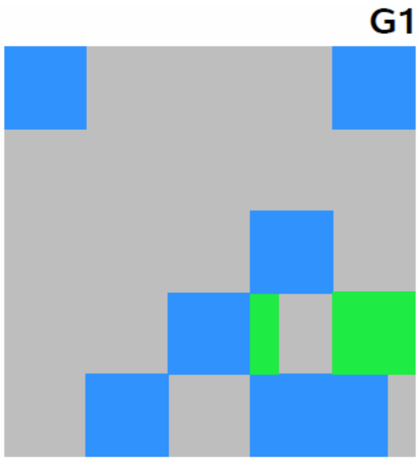



- G1 uses **Remembered Set** (RS) monitoring **cross region references** – ignore inter-region and null references
  - mechanism based on memory barrier for modification of object reference
  - 512 bytes cards in each regions with corresponding dirty flag for each region

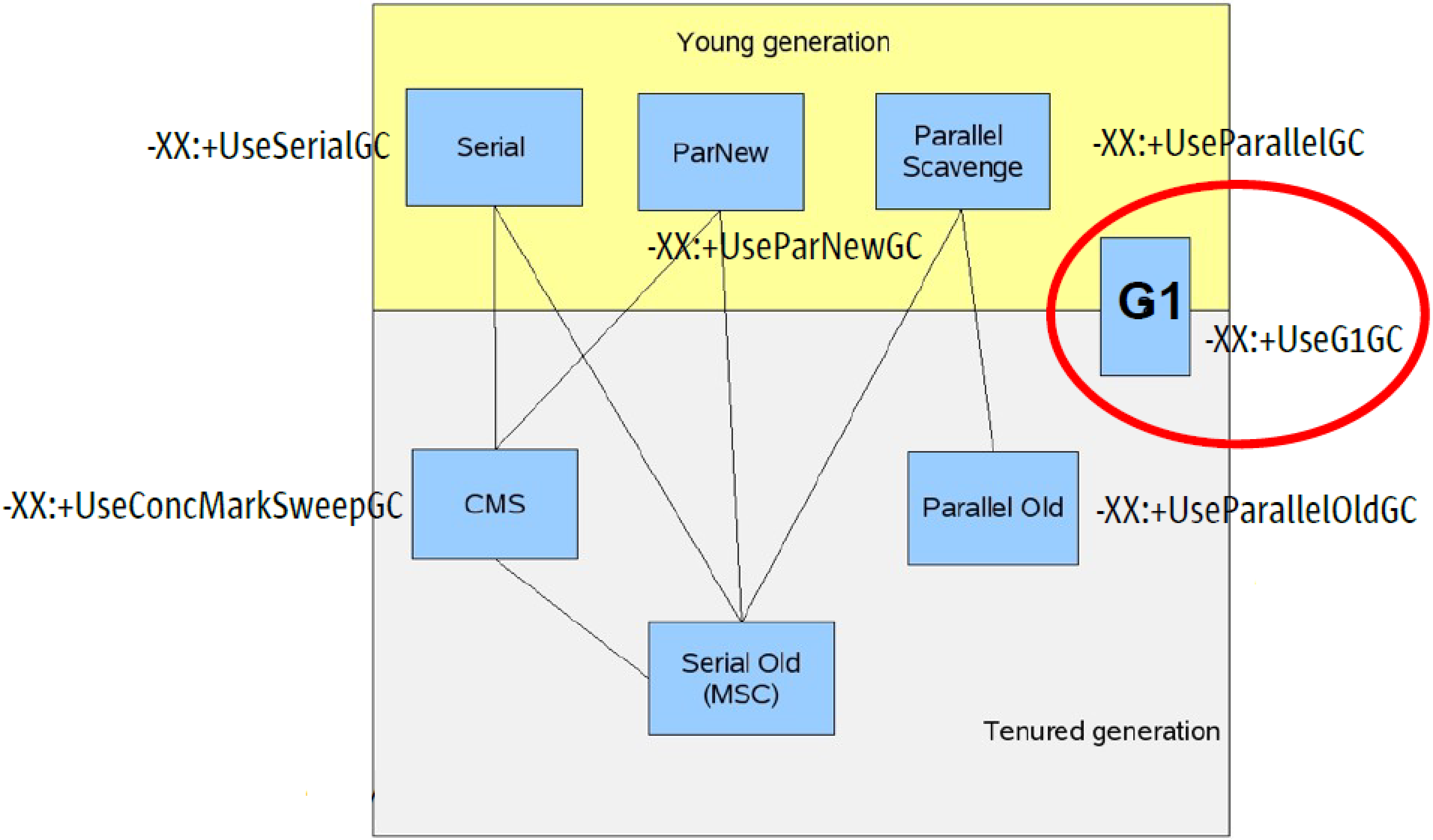
# mixed G1 collector – major GC

» combination of CMS and parallel compacting collector
» runs immediately after minor GC if heap occupancy threshold is met
  -XX:InitiatingHeapOccupancyPercent=n (defualt 45%)
  - initial mark based on SATB (snapshot-at-the-beginning)
    – stop-the-world
  - concurrent marking and region-based stats generation
  - remark
    – stop-the-world
    – reclaim empty regions
  - reclaim old regions (no sweeping using regions)
    – pick regions with low live ratio
    – only few are collected per such GC based on
      -XX:MaxGCPauseMillis=n (default 200ms)
    – leave garbage in regions with high live ratio



# Garbage collectors relation



# Autoboxing, Unboxing

- automatic conversion from primitive to object representation and vice versa
- since JAVA 5
- for example
  - autoboxing for Integer is based on valueOf(int) and intValue() methods

```
int myInt = 3;
myInt.toString();
```

# Autoboxing, Unboxing

- automatic conversion from primitive to object representation and vice versa
- since JAVA 5
- for example
  - autoboxing for Integer is based on valueOf(int) and intValue() methods

```
int myInt = 3;
myInt.toString();
```

- works only during assignment or parameter passing

```
String a = myInt+"";
```

# Autoboxing, Unboxing

- automatic conversion from primitive to object representation and vice versa
- since JAVA 5
- for example
  - autoboxing for Integer is based on valueOf(int) and intValue() methods

```
int myInt = 3;
myInt.toString();
```

- works only during assignment or parameter passing

```
String a = myInt+"";
```

```
Integer.toString(myInt);
```

- example: count word frequency/histogram

```
public static void main(String[] args) {
    Map<String, Integer> m = new TreeMap<String, Integer>();
    for (String word : args) {
        Integer freq = m.get(word);
        m.put(word, (freq == null ? 1 : freq + 1));
    }
    System.out.println(m);
}
```

- boxing and un-boxing brings inefficiencies !

# Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

# Example

```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

» what is the output? and what is the output for i=2000 and j=2000 ?

| | |
|---|---|
| true | true |
| true | false |
| true | true |

» but not after serialization, there is no readResolve !

# Performance recommendations

- » prefer short-lived immutable objects instead of long-lived mutable objects
- » avoid needless allocations
  - more frequent allocations will cause more frequent GCs
- » large objects:
  - expensive to allocate (not in TLAB, not in young)
  - expensive to initialize (zeroing)
  - can cause performance issues
  - fragmentation for CMS (non-compacting) GC
- » avoid force System.gc() except well-defined application phases
  - can be ignored by `-XX:+DisableExplicitGC`
- » avoid frequent array-based re-sizing
  - several allocations
  - a lot of array copying
  - use:

```
ArrayList<String> list = new ArrayList<String>(1024);
```

# Performance recommendations

» avoid **finalizable** objects (non-trivial finalize() method)
  - slower allocation due to their tracking
  - require at least **two GC cycles**:
    - enqueues object on finalization queue
    - reclaims space after finalize() completes
  - beware of extending objects which define finalizers
    - use reference instead of extending
    - manual nulling

# Performance recommendations

» use lazy initialization

```
class Foo {
    private String[] names;
    public void doIt(int length) {
        if (names == null || names.length < length)
            names = new String[length];
        populate(names);
        print(names);
    }
}
```

# Performance recommendations

» objects in the wrong scope

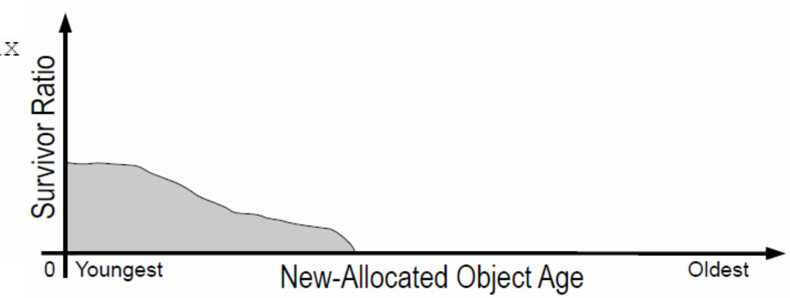
```
class Foo {
    private String[] names;
    public void doIt(int length) {
        if (names == null || names.length < length)
            names = new String[length];
        populate(names);
        print(names);
    }
}
```

```
class Foo {
    public void doIt(int length) {
        String[] names = new String[length];
        populate(names);
        print(names);
    }
}
```

# Performance recommendations

- instances of inner classes have an **implicit reference** to the outer instance
- larger heap space for both generations -> less frequent GCs, lower GC overhead, objects more likely to become dead (smaller heap -> fast collection)
- tune size of young generation -> implies frequency of minor GCs, maximize the number of objects released in young generation, it is better to copy more than promote more
- tune tenuring distribution (-XX:+PrintTenuringDistribution),

```
Desired survivor size 6684672 bytes, new threshold 8 (max
- age   1:    2315488 bytes,    2315488 total
- age   2:      19528 bytes,    2335016 total
- age   3:         96 bytes,    2335112 total
- age   4:         32 bytes,    2335144 total
```




- overall application footprint should not exceed physical memory !
- different Xms and Xmx implies full GC during resizing (consider Xms=Xmx)