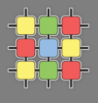
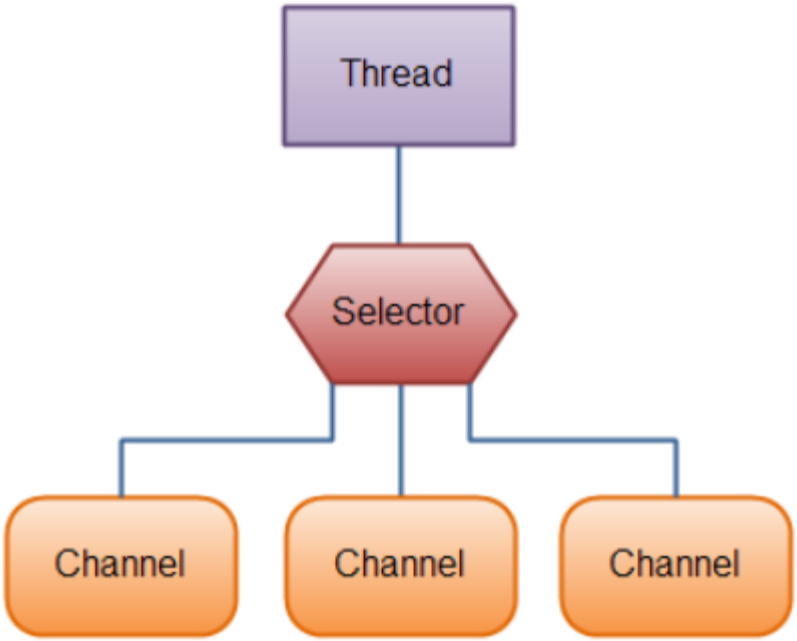


Architecture of software systems

Course 13: NIO networking, Data structures, memory management with garbage collector

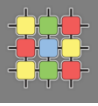
David Šišlák
david.sislak@fel.cvut.cz

JAVA networking – NIO – Channel, Selector



- one thread works with multiple channels at the same time
- Channel – cover UDP+TCP network IO, file IO
  - FileChannel – from Input/OutputStream or RandomAccessFile
  - DatagramChannel
  - MulticastChannel (since 1.7)
  - SocketChannel
  - ServerSocketChannel

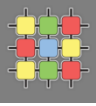
JAVA networking – NIO – Channel



» Channel
  - read/write at the same time (streams are only one-way)
  - always read/write from/to a buffer
  - channel.transferFrom(int pos, int count, Channel source), transferTo …

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "r");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocateDirect(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {

  System.out.println("Read " + bytesRead);
  buf.flip();

  while(buf.hasRemaining()){
      System.out.print((char) buf.get());
  }

  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

JAVA networking – NIO – Selector

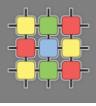


» Selector
  - Selector Selector.open();
  - only channels in non-blocking mode can be registered
    - channel.configureBlocking(false);
    - SelectionKey channel.register(selector, SelectionKey.OP_READ);
  - FileChannel doesn't support non-blocking mode !

» SelectionKey – events you can listen for (can be combined together)
  - OP_CONNECT
  - OP_ACCEPT
  - OP_READ
  - OP_WRITE

» events are filled by channel which is ready with operation

JAVA networking – NIO – Selector



- SelectionKey – returned from register method
  - interest set – your configured ops
  - ready set – which ops are ready, sk.isReadable(), sk.isWritable(), …
  - the channel
  - selector
  - optional attached object – sk.attach(Object obj); Object sk.attachment()
    - SelectionKey channel.register(selector, ops, attachmentObj);

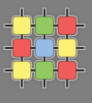
- Selector with registered one or more channels
  - int select() – blocks until at least one channel is ready
  - int select(long timeout) – with timeout milliseconds
  - int selectNow() – doesn't block at all, returns immediately

  - return the number of channels which are ready from the last call !
    - Set<SelectionKey> selector.selectedKeys();

JAVA networking – NIO – Selector

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
}
```

JAVA networking – NIO – Selector, SocketChannel

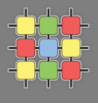


» Selector (cont.)
  - wakeUp() – different thread can “wake up” thread blocked in select()
  - close() – invalidates selector, channels are not closed

» SocketChannel
  - can be configured as non-blocking before connecting
  - SocketChannel socket.getChannel();
  - SocketChannel SocketChannel.open();
  - sch.connect(…)

  - write(…) and read(…) may return without having written/read anything for non-blocking channel !

JAVA networking – NIO

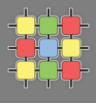


» ServerSocketChannel
  - can be configured as non-blocking
  - can be created directly using open() or from ServerSocket
  - accept() – returns SocketChannel in the same mode

» DatagramChannel
  - can be configured as non-blocking
  - can be created directly using open() or from DatagramSocket
  - receive(…), send(…)

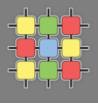
» FileChannel
  - cannot be non-blocking !
  - support – direct buffers, mapped files, locking

Data structures



- primitives: boolean, byte, char, int, long, float, double
  - without implicit allocation
  - placed in frame in variables or operand stack
- objects
  - every object is descendant of Object by default
    - methods – clone(), equals, getClass(), hashCode(), wait(…), notify (…), finalize()
  - objects for primitives: Boolean, Byte, Char, Integer, Long, Float, Double
  - can be null
  - other objects
- arrays
  - special data structure which store a number of items of the same type in linear order; have the defined limit
  - JAVA automatically check limitations
  - allocated on the heap
  - multi-dimensional arrays = arrays of arrays; ragged array

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

NO

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

- works only during assignment or parameter passing

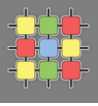
~~`String a = myInt+"";`~~ `Integer.toString(myInt);`

- 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



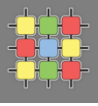
```
int i = 2;
int j = 2;
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(i);
list.add(j);
System.out.printf(Boolean.toString(i==j));
System.out.printf(Boolean.toString(list.get(0)==list.get(1)));
System.out.printf(Boolean.toString(list.get(0).equals(list.get(1))));
```

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

true	true
true	false
true	true

» but not after serialization, there is no readResolve !

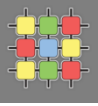
Integer – usage of identity semantics



» similar concept as in multiton

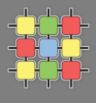
```
// set from vm option -XX:AutoBoxCacheMax=<size>
private static String integerCacheHighPropValue;

private static class IntegerCache {
    static final int high;
    static final Integer cache[];

    static {
        final int low = -128;

        int h = 127;
        if (integerCacheHighPropValue != null) {
            int i = Long.decode(integerCacheHighPropValue).intValue();
            i = Math.max(i, 127);
            h = Math.min(i, Integer.MAX_VALUE - -low);
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);
    }

    private IntegerCache() {}
}
```

Integer



```
public static class Integer {
    private final int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }

    public static Integer valueOf(int i) {
        if(i >= -128 && i <= IntegerCache.high)
            return IntegerCache.cache[i + 128];
        else
            return new Integer(i);
    }
}
```

Widening vs. autoboxing



```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

```
public static void hello(Integer x) {
    System.out.println("Integer");
}

public static void hello(Long x) {
    System.out.println("long");
}

public static void main(String[] args) {
    int i = 5;
    hello(i);
}
```

» what are the outputs?

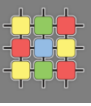
long **Integer**

» why?

prefer widening before autoboxing

cannot use autoboxing to widen primitives -> error if no hello(Integer) method

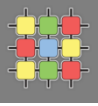
Example



```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove(i - 1);
        }
        System.out.println(s.size());
    }
}
```

» what is the outputs?

Example



```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove(i - 1);
        }
        System.out.println(s.size());
    }
}
```

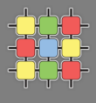
» what is the outputs?

100 - because we are removing Integers instead of Short !!

» correct:

```
public static class ShortSet {
    public static void main(String args[]) {
        Set<Short> s = new HashSet<Short>();
        for (short i = 0; i < 100; i++) {
            s.add(i);
            s.remove((short) (i - 1));
        }
        System.out.println(s.size());
    }
}
```

Method overloading



» method is identified by its signature

```
public static void method(Object obj) {
    System.out.println("method with param type - Object");
}

public static void method(String obj) {
    System.out.println("method with param type - String");
}

public static void main(String[] args) {
    method(null);
}
```

» can be compiled and what is the output?

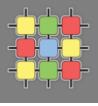
- **YES** – no ambiguity

method with parameter type – String

» due to JLS specification:

"The Java programming language uses the rule that the *most specific* method is chosen."

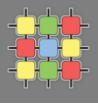
Method overloading



```
public static void method(Object obj){
    System.out.println("method with param type - Object");
}

public static void method(String str){
    System.out.println("method with param type - String");
}

public static void method(StringBuffer strBuf){
    System.out.println("method with param type - StringBuffer");
}
public static void main(String[] args) {
    method(null);
}
```

- can be compiled and what is the output?
  - **NO** – cannot find “most specific”, both are sub-classes of Object but not in the same inheritance hierarchy

Method overloading



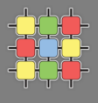
```
public static void method(Object obj, Object obj1) {
    System.out.println("method with param types - Object, Object");
}

public static void method(String str, Object obj) {
    System.out.println("method with param types - String, Object");
}

public static void main(String[] args) {
    method(null, null);
}
```

» can be compiled and what is the output?

- **YES**

method with param types – String, Object

Method overloading

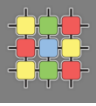


» BUT

```
public static void method(Object obj, String obj1) {
    System.out.println("method with param types - Object, String");
}

public static void method(String str, Object obj) {
    System.out.println("method with param types - String, Object");
}
```

» this cannot be compiled – cannot identify “most specific”

Method overloading



```
public static void hello(Collection x) {
    System.out.println("Collection");
}

public static void hello(List x) {
    System.out.println("List");
}

public static void main(String[] args) {
    Collection col = new ArrayList();
    hello(col);
}
```

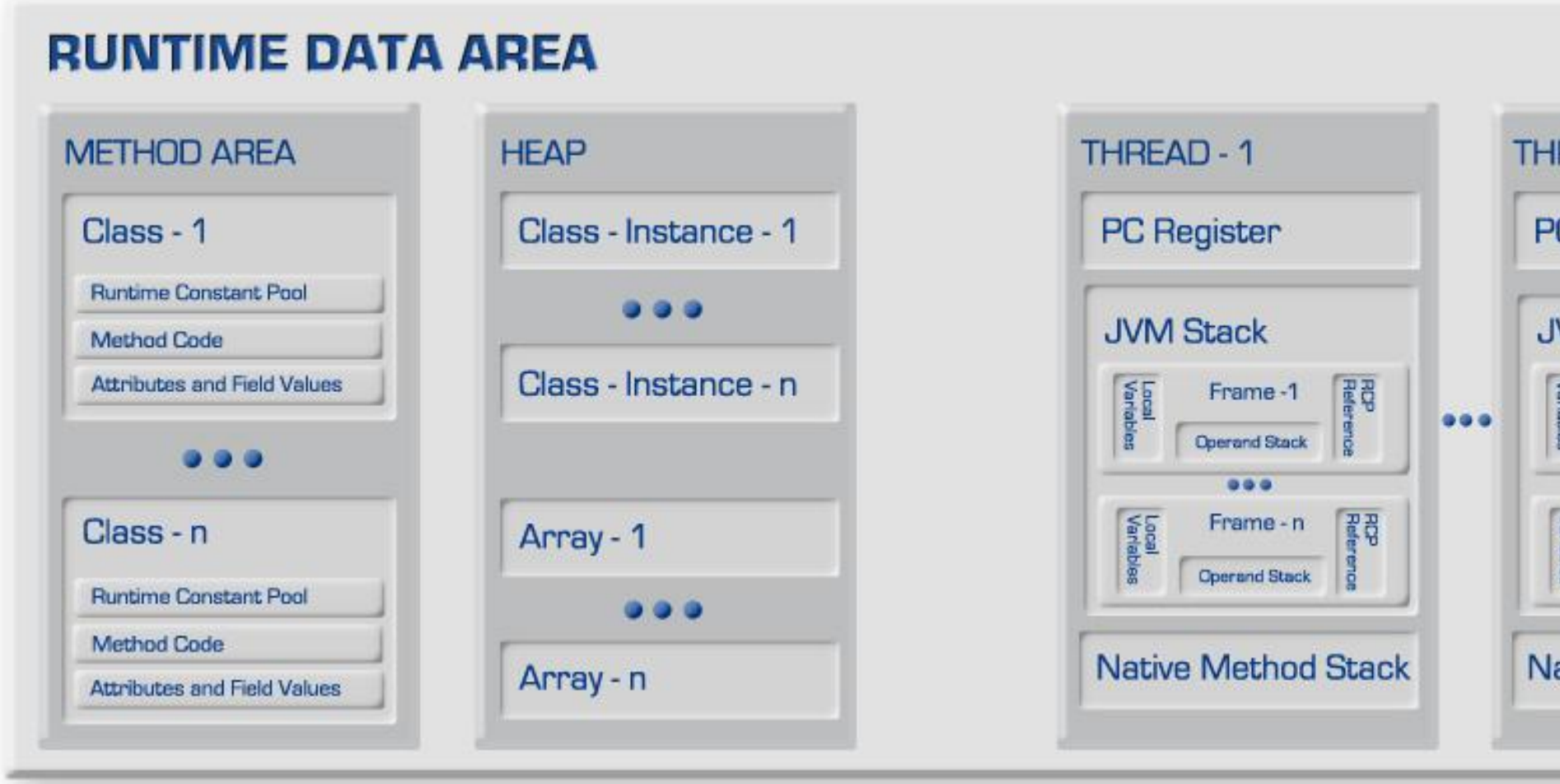
» can be compiled and what is the output?

- **YES**

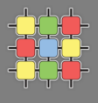
Collection

- compile time resolution not run-time type

JVM Memory

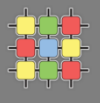

RUNTIME DATA AREA
METHOD AREA
Class - 1
Runtime Constant Pool
Method Code
Attributes and Field Values
Class - n
Runtime Constant Pool
Method Code
Attributes and Field Values
HEAP
Class - Instance - 1
Class - Instance - n
Array - 1
Array - n
THREAD - 1
PC Register
JVM Stack
Local Variables
Frame -1
Operand Stack
RCP Reference
Local Variables
Frame - n
Operand Stack
RCP Reference
Native Method Stack

Memory management

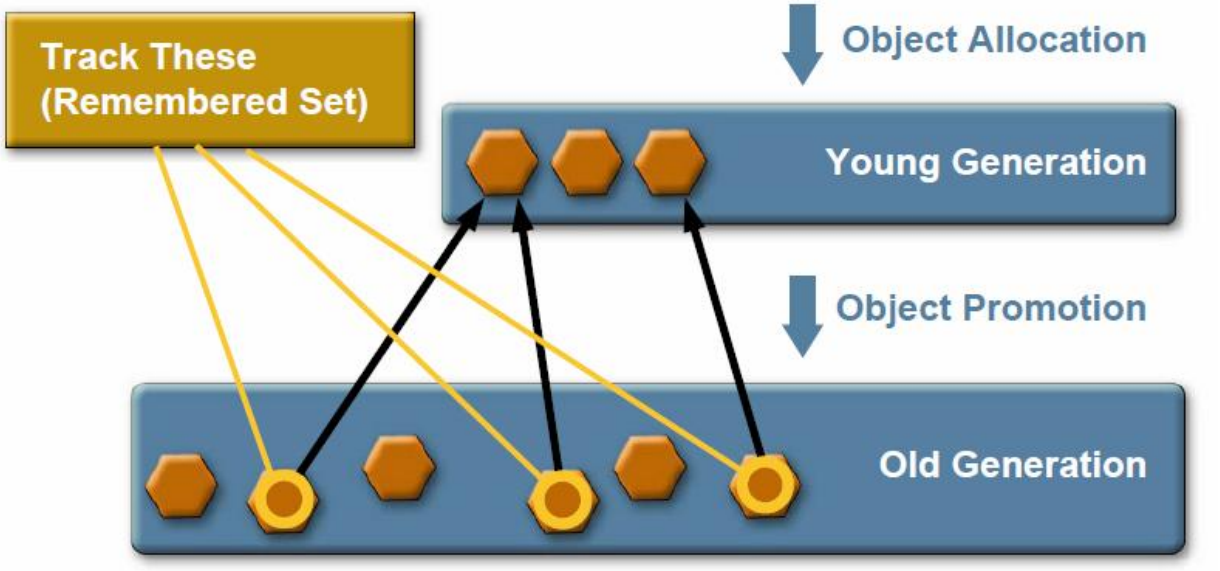


- explicit vs. *automatic*
  - no crashes due to errors – e.g. usage of de-allocated objects
  - no space leaks
- garbage collection managed by *garbage collector*
  - live objects (transiently reachable from **roots** – thread frames, static fields) remain in memory
  - dead are reclaimed
- desired characteristics:
  - allocation performance - find a block of unused memory with certain size
  - avoid fragmentation (e.g. by compaction)
  - efficiency without long pauses in application run
  - no bottleneck for multi-threaded (multi-CPUs) systems
- design architectures:
  - serial vs. parallel
  - concurrent vs. stop-the-world
  - compacting vs. non-compacting vs. copying

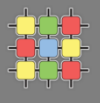
Generational concept



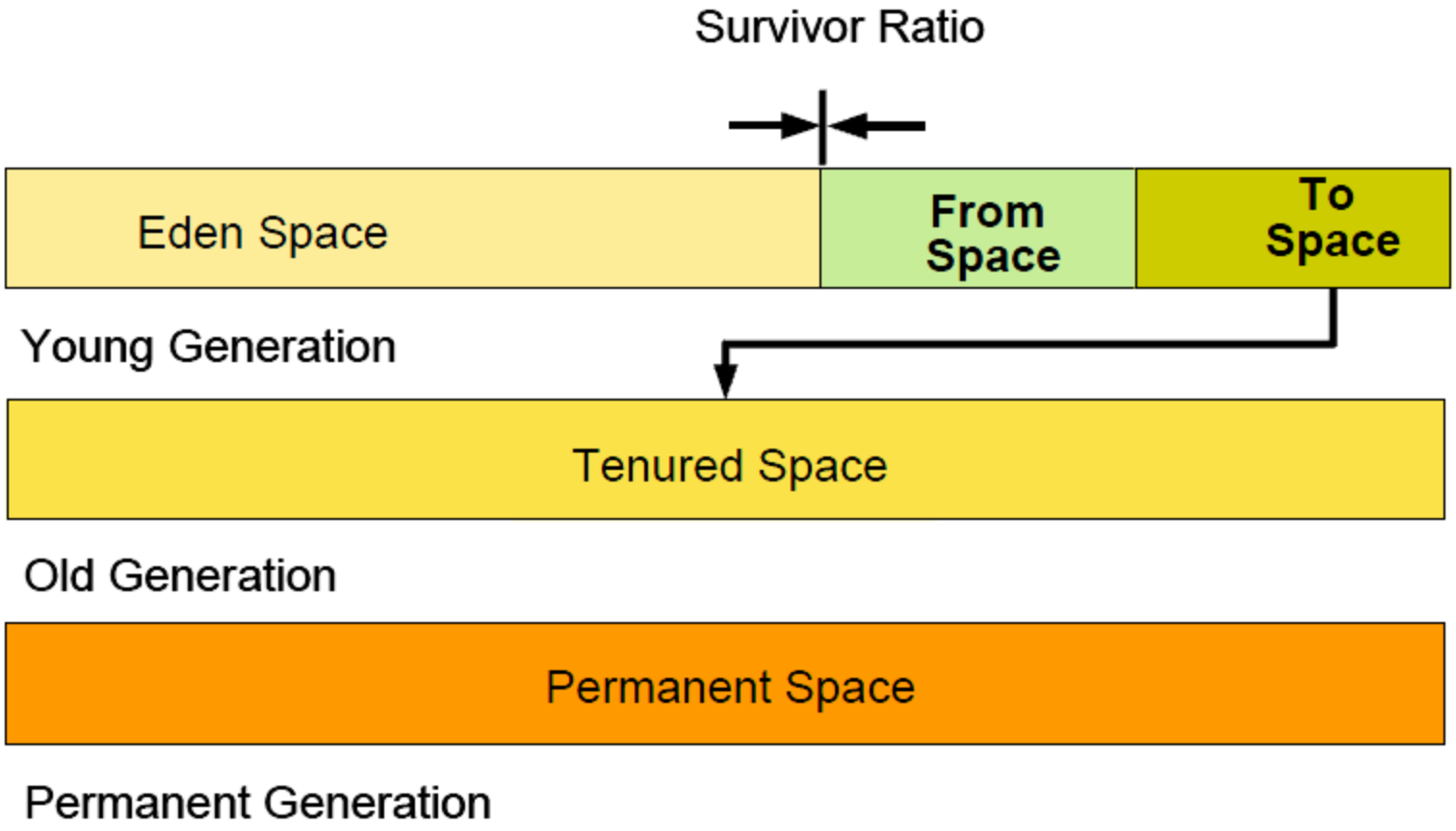
- heap divided into generations based on object ages:
  - young – frequent GC, small size -> fast GC
  - old – rare GC, large size -> slow GC
- promotion (tenuring) objects based on survival of objects during GC
- based on weak generational hypothesis:
  - most allocated objects are not referenced for long – they die young
  - few references from older to younger object exist
- need track old-to-young references



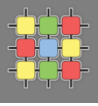
JAVA heap layout



- *minor (young)* vs. *major (old)* GC – different algorithms
- major GC can be invoked by young GC if there is no space in tenured space

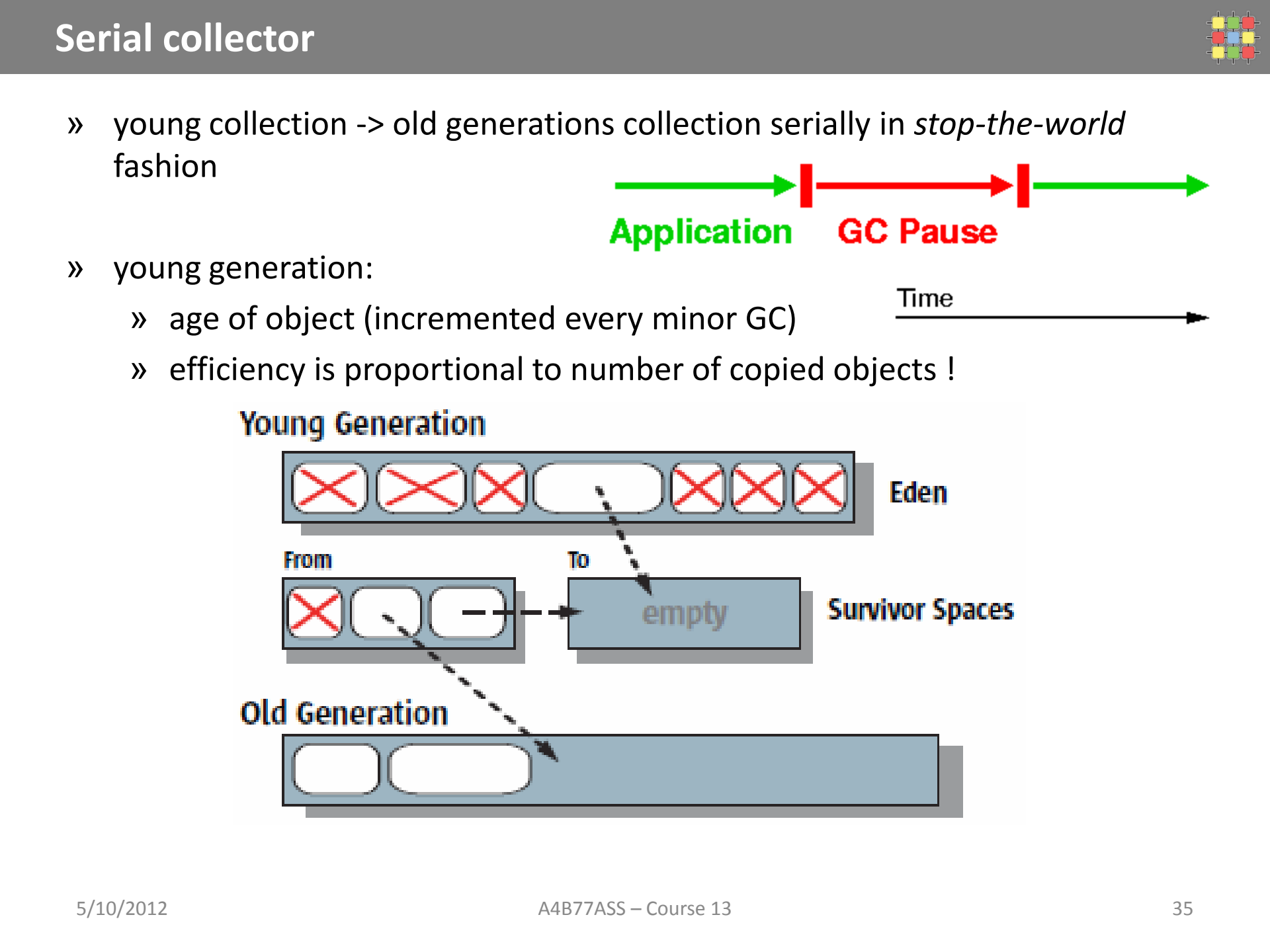


Fast allocation



- based on *bump-the-pointer* technique
  - track previously allocated object
  - fit new object into remainder of generation end
- thread-local allocation buffers (TLABs)
  - remove concurrency bottleneck
  - each thread has very small exclusive area (about 1% of Eden in total)
  - infrequent full TLABs implies synchronization (based on CAS)
  - exclusive allocation takes about *10 native instructions*

Serial collector

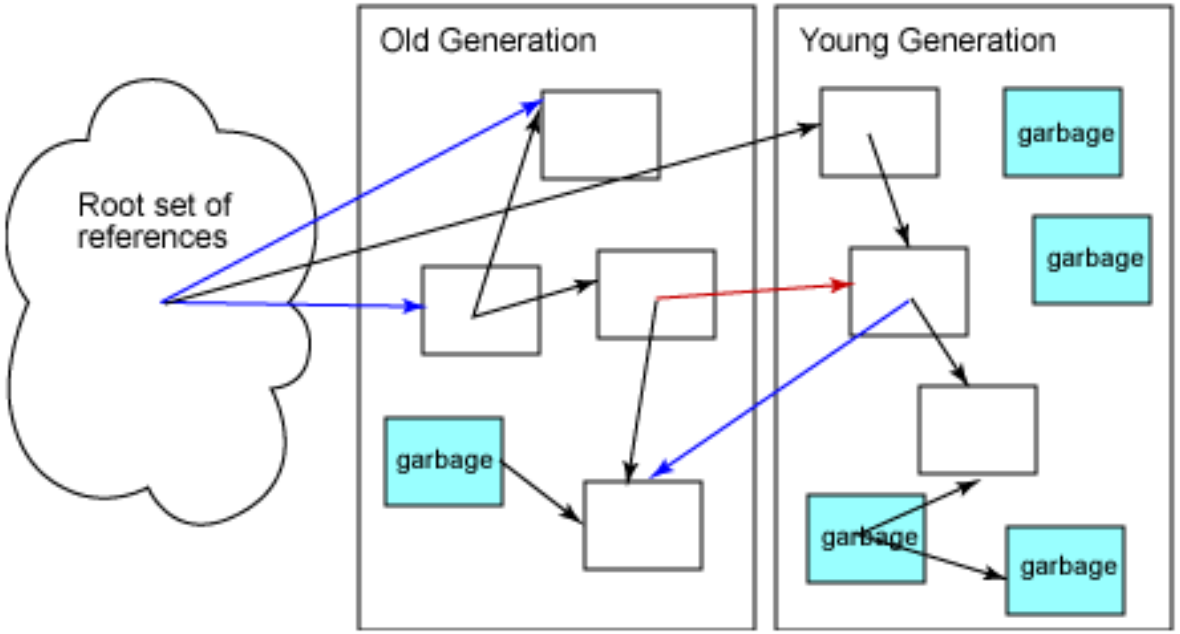


- young collection -> old generations collection serially in *stop-the-world* fashion

- young generation:
  - age of object (incremented every minor GC)
  - efficiency is proportional to number of copied objects !

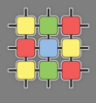
Young generation live object detection – IBM version

- maintains separate list of old-to-young references as they are created
- maintain the list during object promotion, introduce new, remove old

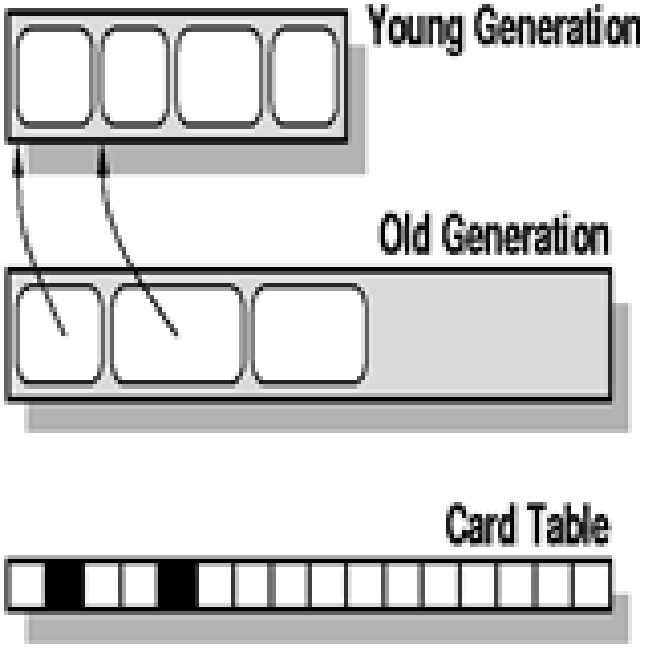


red – old-to-young, blue – to old (don't need trace during minor collection)

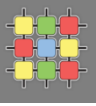
Young generation live object detection – Sun version



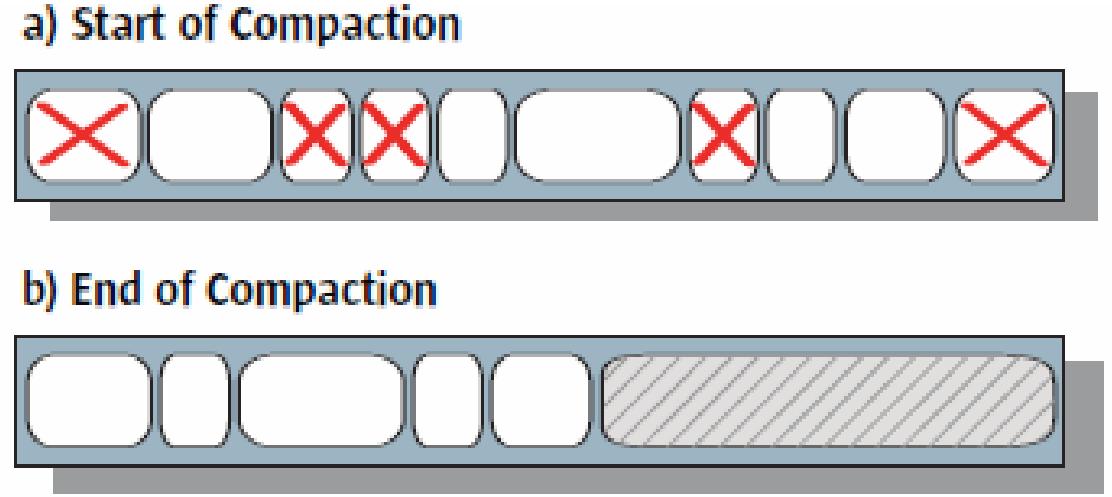
- identification of live objects based on card table structure (boolean)
- 512-byte chunks in old generation (smaller than memory page)
- every update to a reference marks dirty
- bytecode interpreter and JIT uses reference write barrier to maintain card table
- only dirty cards are scanned for old-to-young references
- finally marks are cleared



Serial collector



» old and permanent generation:
  - using ***mark-sweep-compact*** algorithm
  - allocation can use *bump-the-pointer* technique



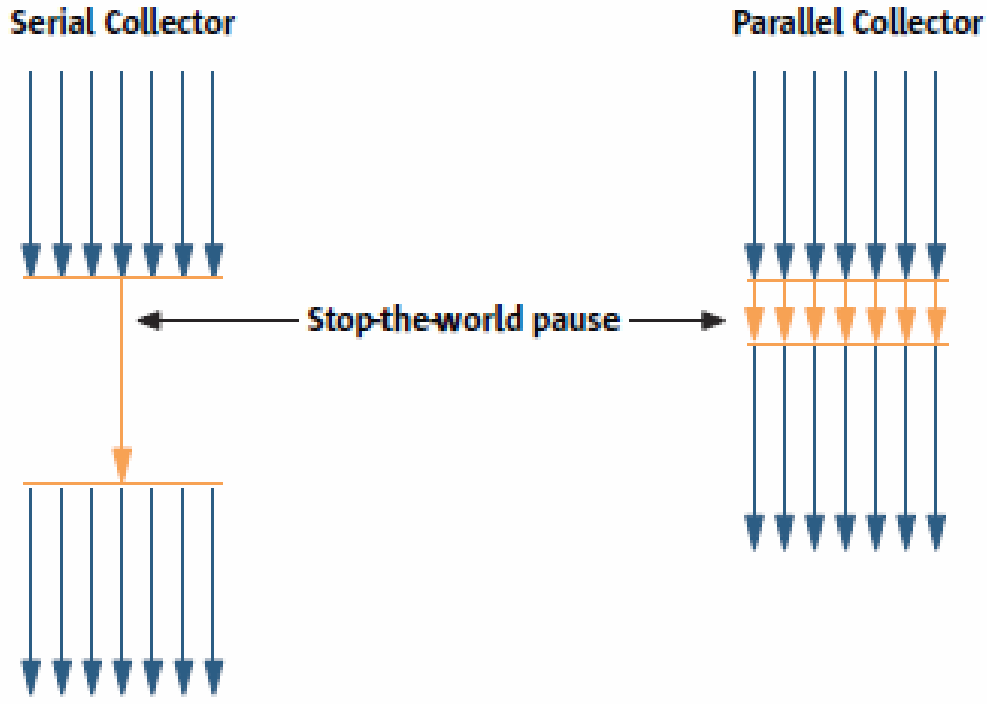
» default in Java 5.0 for client JVM

» effectively handles application with 64MB heaps

» `-XX:+UseSerialGC`

Parallel collector

- utilize more cores/CPUs
- still stop-the-world **but in parallel** manner for **young generation**

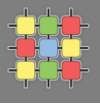


- **uses the same serial *mark-sweep-compact* algorithm for old generation**
- default for server JVM from Java 5.0
- `-XX:+UseParallelGC`

Parallel compacting collector

- introduced in J2SE 5.0 update 6
- no change in **young generation collection – use parallel one**
- old and permanent generations:
  - done in stop-the-world manner
  - each generation logically divided into fixed-sized regions
  - ***parallel mark*** phase:
    - initiated by divided reachable live objects
    - info about live objects (size & location) are propagated to the corresponding region data

Parallel compacting collector

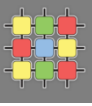


- ***summary phase*** (implemented in serial):
  - identify density of regions (due to previous compactions, more dense are at the beginning)
  - find from which region it has sense to do compaction regarding recovered from a region:
    - » *dense prefix* – before, no movement
  - calculate new location of each live data for each region
- ***compaction*** phase:
  - parallel copy of data based on the summary data
  - finally heap is packed and large empty block is at the end

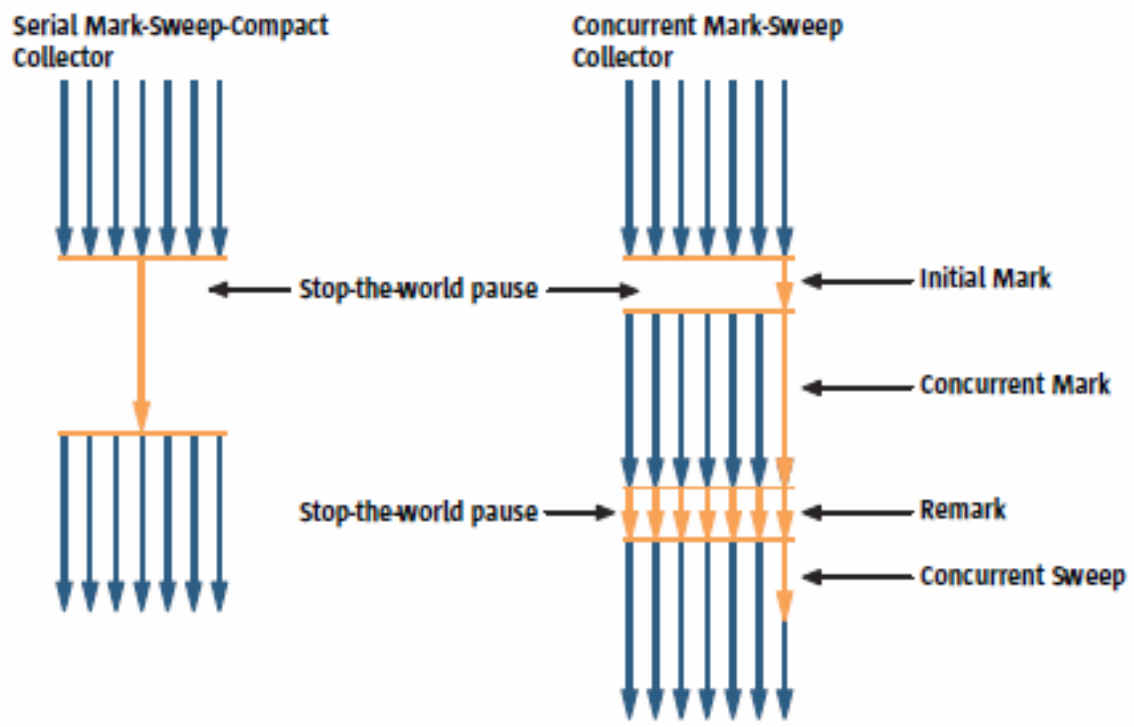
» `-XX:+UseParallelOldGC,-XX:ParallelGCThreads=n`

» default in J2SE 6.0 for multi core/CPU systems

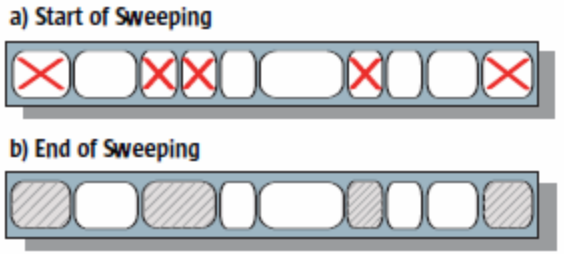
Concurrent mark-sweep (CMS) collector



- low-latency collector
- use the same parallelized young generation collector
- old generation:
  - done concurrently with the application execution
  - *initial mark* – short pause identifying the initial set of live objects directly reachable
  - *remark* – revisiting modified objects (overhead)
  - *concurrent sweep*



Concurrent mark-sweep (CMS) collector

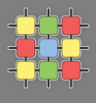


- non-compacting
- cannot use bump-the-pointer
- more expensive allocation searching a region
- extra overhead to young generation collection doing promotions
- may split or join free block depending on tracked popular object sizes
- collector started:
  - adaptively based on previous runs (how long it takes, how many is free)
  - initiating occupancy in percentage

    `-XX:CMSInitiatingOccupancyFraction=n`

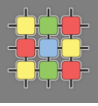
    default 68
- decreases pauses
- requires larger heap due to concurrent collection
- *incremental mode* – concurrent phases divided into small chunks between young generation collection
- `-XX:+UseConcMarkSweepGC , -XX:+CMSIncrementalMode`

Configure garbage collector



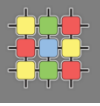
- explicit type:
  - `-XX:+UseSerialGC, -XX:+UseParallelGC, -XX:+UseParallelOldGC, -XX:+UseConcMarkSweepGC`
- statistics:
  - `-XX:+PrintGC, -XX:+PrintGCDetails, -XX:+PrintGCTimeStamps, -XX:+PrintTenuringDistribution`
- heap sizing:
  - `-Xmx` – max heap size, default 64MB on client JVM, influence to throughput
  - `-Xms` - initial heap size
  - `-XX:MinHeapFreeRatio=min` – default 40, per generation
  - `-XX:MaxHeapFreeRatio=max` – default 70
  - `-XX:NewSize=n` - initial size of young generation
  - `-XX:MaxNewSize=n`

Configure garbage collector



- heap sizing cont.:
  - `-XX:NewRatio=n` - ratio between young and old gens
    default 2 client JVM, 8 server JVM (young includes survivor),
    n=2 => 1:2 => young is 1/3 of total heap
  - `-XX:SurvivorRatio=n` – ratio between each survivor and Eden
    default 32, n=32 => 1:32 => each survivor is 1/34 of young size
  - `-XX:MaxTenuringThreshold=<threshold>`
  - `-XX:PermSize=n` - initial size of permanent generation
  - `-XX:MaxPermSize=n` - max size of permanent generation
- parallel collector & parallel compacting collector:
  - `-XX:ParallelGCThreads=n` - number of GC threads
  - `-XX:MaxGCPauseMillis=n` - maximum pause time goal
  - `-XX:GCTimeRatio=n` - throughput goal
    1/(1-n) percentage of total time for GC, default n=99 (1%)

Configure garbage collector



- CMS collector:
  - `-XX:+CMSIncrementalMode` – default disabled
  - `-XX:ParallelGCThreads=n`
  - `-XX:CMSInitiatingOccupancyFraction=<percent>`
  - `-XX:+UseCMSInitiatingOccupancyOnly` - disable automatic initiating occupancy
  - `-XX:+CMSClassUnloadingEnabled` - by default disabled !!!
  - `-XX:CMSInitiatingPermOccupancyFraction=<percent>` - unloading has to be enabled !!!
  - `-XX:+ExplicitGCInvokesConcurrent`
  - `-XX:+ExplicitGCInvokesConcurrentAndUnloadClasses` - both useful when want to references / finalizers to be processed