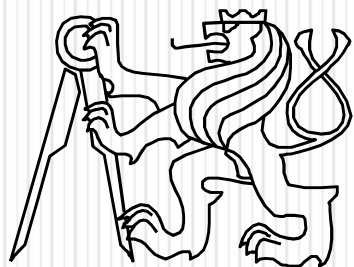


Java Performance

Zdeněk Troníček

Faculty of Information Technology
Czech Technical University in Prague
Czech Republic



tronicek@fit.cvut.cz
<http://users.fit.cvut.cz/~tronicek>

© Zdeněk Troníček, 2011

Who am I?

Zdeněk Troníček, tronicek@fit.cvut.cz

- Assistant professor at Faculty of Information Technology, Czech Technical University in Prague
- Teaching courses Enterprise Java, Programming in Java, Programming languages and Compilers, Database systems
- 10+ years experience with Java trainings
- Certifications:
 - Sun Certified Java Programmer
 - Sun Certified Web Component Developer
 - Sun Certified Business Component Developer
 - Sun Certified Specialist for NetBeans IDE
- Blog <http://tronicek.blogspot.com/>

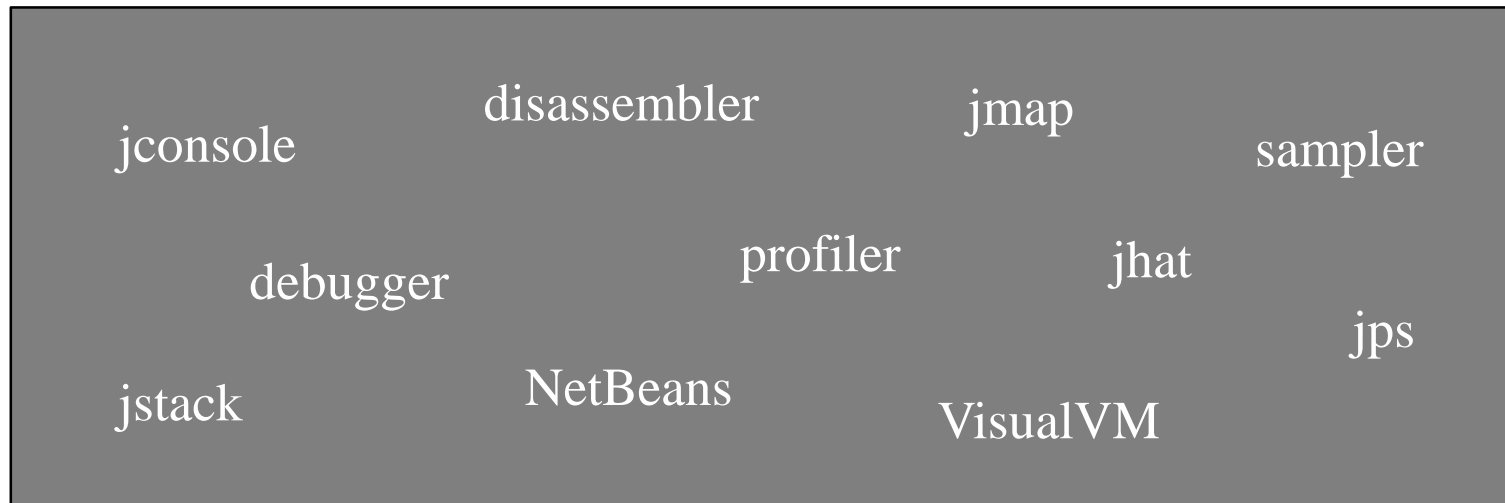
You will learn...

- How to approach performance problems
- How to find a bottleneck in Java program
- How to solve some common problems
- Something about how Java Virtual Machine (JVM) works

Warning!

- This course does not teach tools
- This course does not contain any performance tips you should follow

Tools



You can use any tool, provided that it gives information you need.

Motto: Tools do not matter. Methodology does.

Prerequisites

- Enthusiasm
- Knowledge of the Java programming language (we will not do any programming but we will read source codes)

Course Map

- Introduction
- Java Virtual Machine
- Memory management
- Threads and synchronization

Introductions

- Name
- Affiliation
- Experience related to topics presented in this course
- Expectations for this course
- Something interesting about your country

Questions & Answers

Next module: Introduction

1 Introduction

Module Content

- Performance improvements
- Premature optimization
- Coding in Java
- Hardware monitoring
- Performance patterns

Introduction

- Our goal is to improve user experience
- Java performance tuning is more an art than a science
- Do not optimize prematurely
- Correctness is always over performance

Kirk Pepperdine: “Measure, don’t guess”

User Experience

- User experience is subjective
- Focus on improving the user experience
- User experience can be improved by
 - optimization
 - tuning
 - other means (e.g. a progress bar)

Demo: User Experience

1. Run the project in demo/Introduction/UserExperience
2. Compare your perception of operation time if the operation is being performed with and without a progress bar

Performance Tips

- Do not trust them
- Do not use them unless they are justified by measurement
- Beware of premature optimization

Premature Optimization

D. E. Knuth (1974): “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

M. A. Jackson (1975): “We follow two rules in the matter of optimizations:

Rule 1. Don't do it.

Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution.”

Why is premature optimization bad?

- It seldom has any effect on performance
- It usually worsens readability

Premature Optimization (cont.)

Let's assume that findMax is $O(m)$...

```
for ( int i : values ) {  
    if ( i > findMax ( anotherValues ) ) { ... }  
}
```

$O(m * n)$ vs. $O(m + n)$

```
int max = findMax ( anotherValues );  
for ( int i : values ) {  
    if ( i > max ) { ... }  
}
```


Premature Optimization (cont.)

```
for ( int i = 0, n = p.size(); i < n; i++ ) { ... }
```

vs.

```
for ( int i = 0; i < p.size(); i++ ) { ... }
```

This is okay if `size()` is $O(1)$

How to Code in Java?

- Focus on architecture
- Use efficient technologies and frameworks
- Use sound design principles and patterns
- Use efficient algorithms
- Prefer readable and understandable code

Brian Goetz: “Write dumb code”

Performance Tuning

- The goal of performance tuning is better (subjective) performance of application
- How to arrange it?
 - Optimize the application
 - Increase hardware utilization
 - Tune JVM (garbage collector, JIT, ...)
 - Use other means (progress bar, ...)
 - Use more efficient hardware

Performance Tuning (cont.)

Performance problem can be in

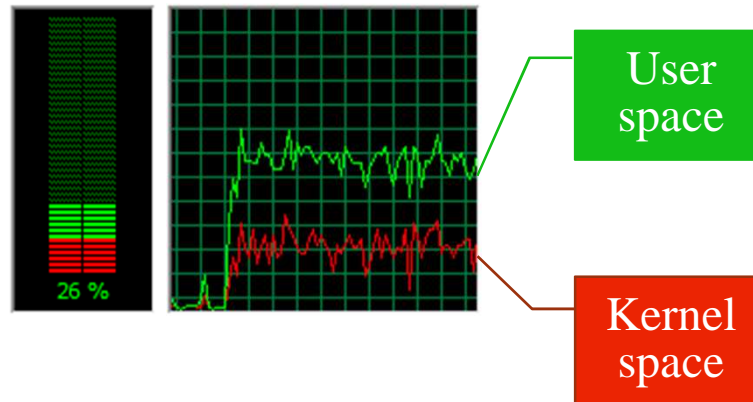
- Hardware
 - CPU
 - Memory
 - I/O (disk, network)
- JVM configuration
- Application

Hardware Monitoring

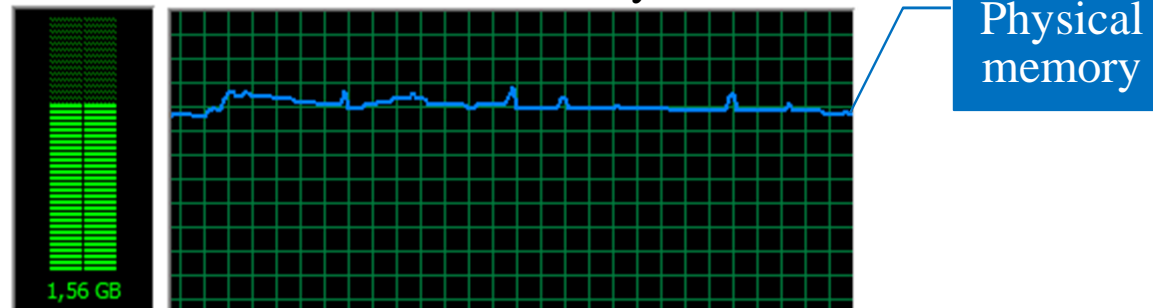
Tools

- Task Manager
- vmstat

Utilization of CPU

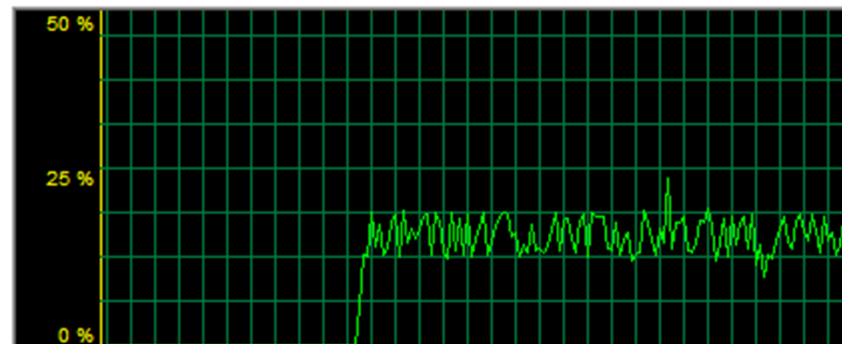


Utilization of memory

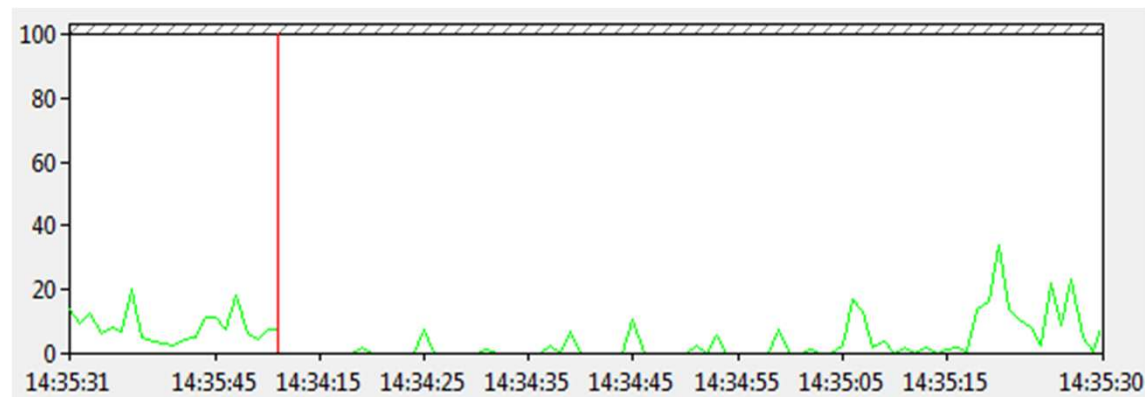


Hardware Monitoring (cont.)

Utilization of network



Utilization of disk



vmstat procs

procs			memory		page						disk				faults			cpu			
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

r = runnable threads in run queue

b = blocked threads

w = runnable but swapped

vmstat memory

procs			memory		page							disk				faults			cpu		
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

swap = swap space available (in kilobytes)

free = memory available (in kilobytes)

vmstat page

procs			memory		page							disk				faults			cpu		
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

pi = kilobytes paged in

po = kilobytes paged out

vmstat disk

procs			memory		page							disk				faults			cpu		
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

number of disk operations per second

vmstat faults

procs			memory		page							disk				faults			cpu		
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

in = device interrupts

sy = system calls

cs = context switches

vmstat cpu

procs			memory		page							disk				faults			cpu		
r	b	w	swap	free	re	mf	pi	po	fr	de	sr	s0	s1	s2	s3	in	sy	cs	us	sy	id
0	0	0	11456	4120	1	41	19	1	3	0	2	0	4	0	0	48	112	130	4	14	82
0	0	1	10132	4280	0	4	44	0	0	0	0	0	23	0	0	211	230	144	3	35	62
0	0	1	10132	4616	0	0	20	0	0	0	0	0	19	0	0	150	172	146	3	33	64
0	0	1	10132	5292	0	0	9	0	0	0	0	0	21	0	0	165	105	130	1	21	78

us = user time

sy = system time

id = idle time

Performance Bounds

Performance can be bound by

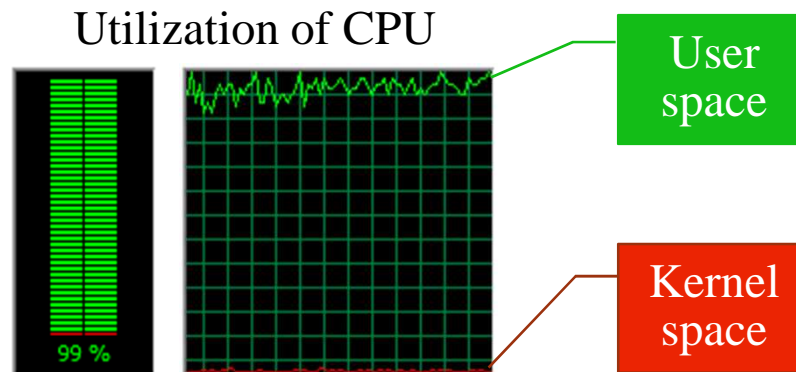
- CPU
- Memory
- I/O (disk or network)
- Locks

These bounds manifest themselves in different ways

Performance Patterns

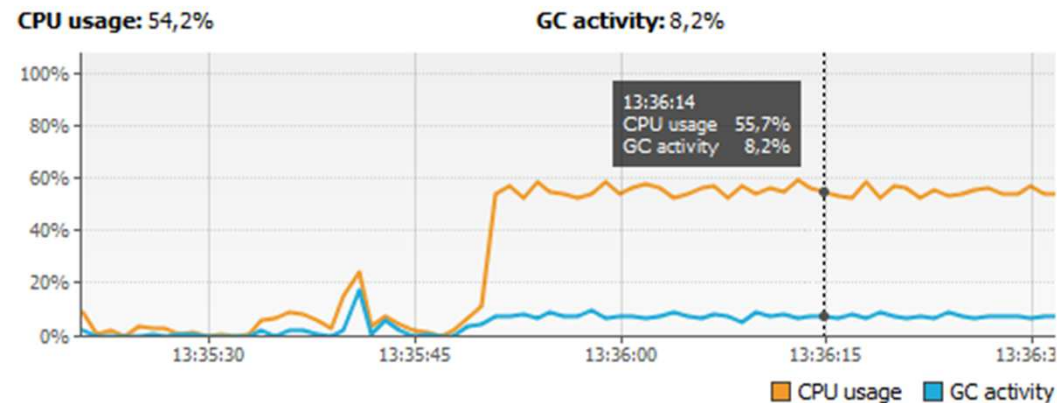
Manifestation	Possible cause
CPU utilization in user space	Application or JVM
CPU utilization in kernel space	Bad interaction with operating system Locks
Idle CPU	Locks Small thread pool Small connection pool Slow I/O

CPU Utilization in User Space



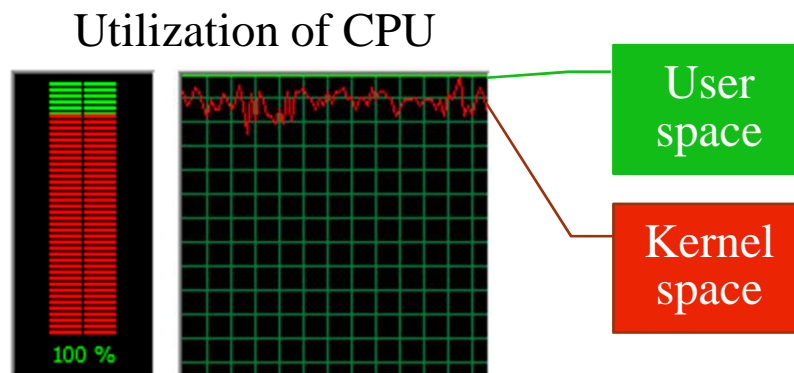
Observation	Next step
CPU utilization is near 100% and mostly in user space	Monitoring garbage collector

GC Monitoring (Visual VM)



Observation	Next step
GC activity is $< 5\%$ of CPU	CPU profiling
GC activity is $> 10\%$ of CPU	Tuning garbage collector

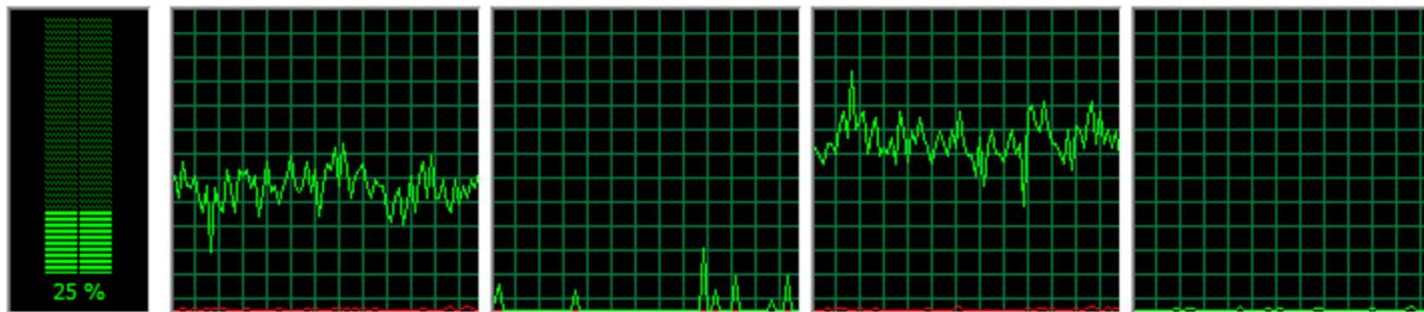
CPU Utilization in Kernel Space



Observation	Next step
CPU utilization in kernel space is more than 10% or more than $\frac{1}{2}$ of total time	CPU profiling

Idle CPU

Utilization of CPU



Observation	Next step
CPU is not fully utilized	Monitoring threads

CPU Profiling

Hot Spots - Method	Self time [%] ▼	Self time	Invocations
java2d.AnimatingSurface. run ()	<div><div></div></div>	225479 ms (38,6%)	18
java2d.MemoryMonitor\$Surface. run ()	<div><div></div></div>	115568 ms (19,8%)	1
java2d.PerformanceMonitor\$Surface. run ()	<div><div></div></div>	114427 ms (19,6%)	1
java2d.demos.Transforms.TransformAnim. render (i...	<div><div></div></div>	92130 ms (15,8%)	7097
java2d.Intro\$Surface\$Scene. pause (Thread)	<div><div></div></div>	10979 ms (1,9%)	17
java2d.Surface. createGraphics2D (int, int, java.a...	<div><div></div></div>	7905 ms (1,4%)	14397
sun.awt.image.ImageFetcher. run ()	<div><div></div></div>	5604 ms (1%)	2
java2d.Surface. paintImmediately (int, int, int, int)	<div><div></div></div>	2474 ms (0,4%)	14348

☰ [Method Name Filter] ▼

Methods with largest time are candidates for optimization

Profiler

- Sampling – the profiler agent periodically samples the stack of all running threads or periodically polls for class histogram
- Instrumentation – the profiler agent instruments class files so that it can, for example, record method invocations

Instrumentation

before

```
public void m1() {  
    m2();  
    m3();  
}
```

after

```
public void m1() {  
    long start = currentTime();  
    try {  
        m2();  
        m3();  
    } finally {  
        long time = currentTime() - start;  
        log( "m1", time );  
    }  
}
```

Instrumentation: systematic error

```
void a() { while (b()) { c(); ... } }  
boolean b() { for (...) { d(); ... } return ... }  
void c() { ... }  
void d() { e(); f(); }  
void e() { ... }  
void f() { ... }
```

Even if we were able to subtract the overhead from the measurements, the code was changed and JVM would behave differently

Sampling vs. Instrumentation

	Sampling	Instrumentation
Code modification	None	Byte-code is modified (JVM does not work the same way)
Overhead	Only when sampling	Large overhead at byte-code modification, small constant overhead while running
Accuracy	Results are analyzed statistically, error ratio is small and decreasing	Produces systematic error that is caused by embedded instructions
Repeatability	Different session might produce different results	Different session produces similar results

Recommendation: start with sampling to get the first idea

Monitoring Threads



- Sleeping: `Thread.sleep(...)`
- Wait: `Object.wait()`
- Monitor: trying to enter a guarded section

Demo: Visual VM

1. Run Visual VM from jdk/bin
2. Open some application in Visual VM
3. Monitor the application

Lab: Monitoring

Classify programs in labs/lab1 into four categories:

- CPU utilized by application
- CPU utilized by garbage collector
- CPU utilized by operating system
- Idle CPU

Conclusion

- Performance improvements
- Premature optimization
- Hardware monitoring
- Performance patterns

Questions & Answers

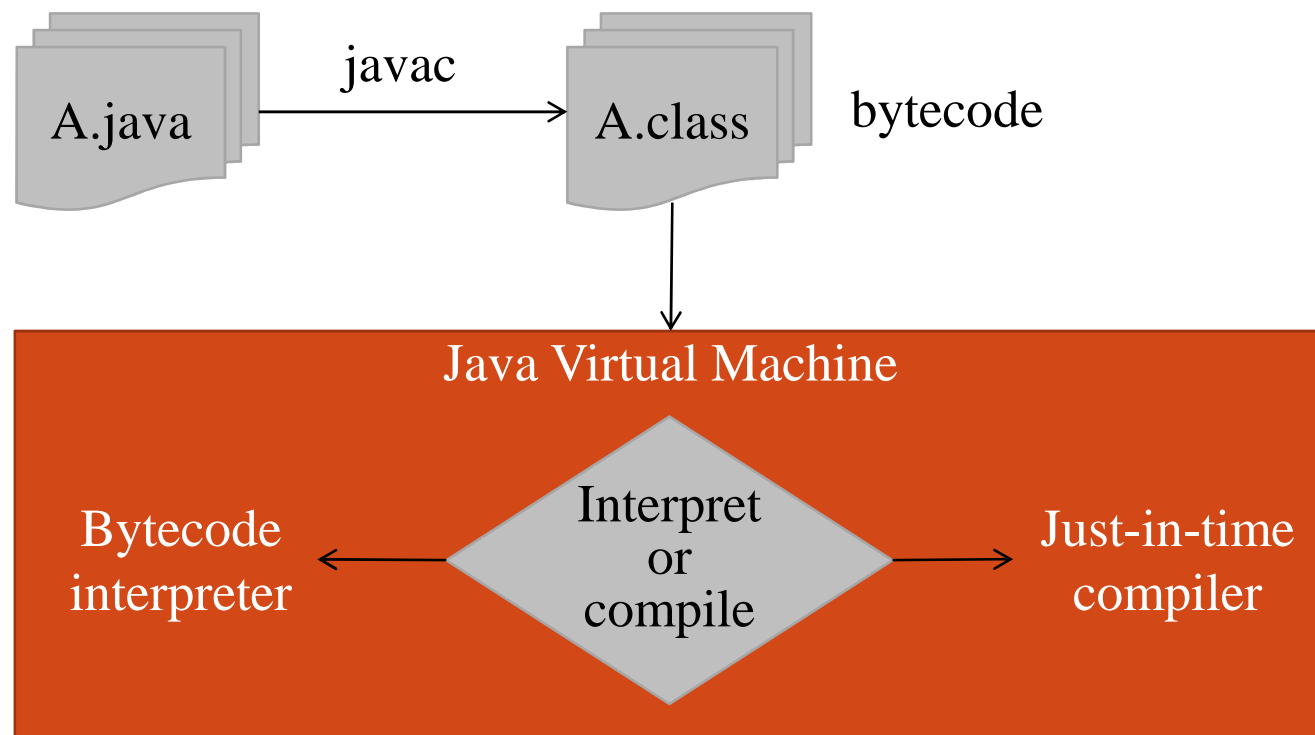
Next module: Java Virtual Machine

2 Java Virtual Machine

Module Content

- Java bytecode
- Java disassembler
- Just-in-Time (JIT) compiler
- Micro-benchmark

Java Virtual Machine (JVM)



Java Bytecode

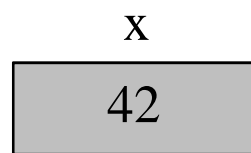
`int x = 42;`

`bipush 42`

`istore_0`

Instructions are typed:

`i` = int, `d` = double, ...



`istore 0`



Operand stack

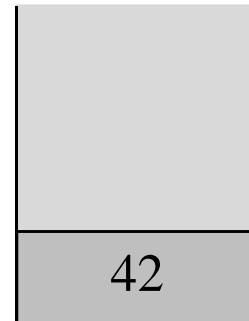


Local variables

x



`bipush 42`



x



Java Bytecode (cont.)

`int y = x + 1;`

`iload_0`
`iconst_1`
`iadd`
`istore_1`

`iload_0`

`0x1a`

`iload 0`

`0x15`

`0x00`

`x`

`42`

`y`

`iload 0`

`iconst_1`

`x`

`42`

`y`

`iadd`

`43`

`x`

`42`

`y`

Class File Structure

- Magic number (0xCAFEBAFE, 4 bytes)
- Minor and major versions of the class file (2 bytes, 2 bytes)
- Constant pool
- Access flags (2 bytes)
- The class name (2 bytes)
- The superclass name (2 bytes)
- Interfaces
- Fields
- Methods
- Attributes

Constant Pool

```
StringBuilder sb =  
    new StringBuilder();  
sb.append( "tortilla" );
```

```
new #1  
dup  
invokespecial #2  
astore_1  
ldc #3  
invokevirtual #4
```

Constant pool

Index	Type	Value
1	Class	#8
2	Method	#1.#7
3	String	#9
4	Method	#1.#12
5	Asciz	<init>
6	Asciz	()V
7	NameAndType	#5:#6
8	Asciz	java/lang/StringBuilder
9	Asciz	tortilla
...

Bytecode Descriptors

Java type	Bytecode descriptor
int	I
long	J
float	F
double	D
char	C

Java type	Bytecode descriptor
boolean	Z
String	Ljava/lang/String;
int[]	[I
int[][]	[[I
void	V

Method	Bytecode descriptor
int m1()	p1/A.m1:()I
void m2(int x)	p1/A.m2:(I)V
String m3(int x, int y)	p1/A.m3:(II)Ljava/lang/String;
void m4(String s)	p1/A.m4:(Ljava/lang/String;)V

Java Disassembler

```
javap <options> <classes>
```

- | | |
|----------|--|
| -c | Disassemble the code |
| -public | Show only public classes and members |
| -private | Show all classes and members |
| -verbose | Print stack size, number of locals and arguments for methods |

Lab: String Concatenation

1. Open the project in labs/lab2/StringConcatenation in NetBeans
2. Get familiar with sources
3. Compile the project
4. Use disassembler to compare the + operator and the StringBuilder class

Evaluation

```
String concat( String s1, String s2 ) {  
    return s1 + " " + s2;  
}
```

These two
compile the
same way

```
String concat( String s1, String s2 ) {  
    return new StringBuilder().append( s1 ).append( " " ).append( s2 ).toString();  
}
```

Evaluation (cont.)

```
String concatNames( String... names ) {  
    String s = "";  
    for ( String n : names ) {  
        s += n + " ";  
    }  
    return s;  
}
```

These two
compile the
same way

```
String concatNames( String... names ) {  
    String s = "";  
    for ( String n : names ) {  
        s = new StringBuilder().append( s ).append( n ).append( " " ).toString();  
    }  
    return s;  
}
```


Discussion

- What does the result say about performance?
- Should we use the + operator or StringBuilder for string concatenation?

Measuring Time

- `System.currentTimeMillis()` – returns the number of milliseconds since January 1, 1970
- `System.nanoTime()` – returns the number of nanoseconds since some fixed but arbitrary time
- `interface ThreadMXBean`
 - `long getCurrentThreadCpuTime()`
 - `long getThreadCpuTime(long id)`

How to measure?

```
long start = System.nanoTime();  
  
// some code  
  
long end = System.nanoTime();  
  
long time = end - start;
```

Both `currentTimeMillis()` and `.nanoTime()` are implemented through JNI and their costs vary from system to system

Resolution

resolution of `System.currentTimeMillis()` is

- 10 ms on Windows NT, Windows 2000
- 15.6 ms on Windows XP and Windows 7
- 1 ms on Linux 2.6, Solaris, and other Unix's

resolution of `System.nanoTime()`

- depends on the operating system
- should not be worse than `System.currentTimeMillis()`
- usually is in microseconds range

ThreadMXBean

`getCurrentThreadCpuTime()` returns the total CPU time for the current thread in nanoseconds

```
ThreadMXBean bean = ManagementFactory.getThreadMXBean();  
  
long time1 = bean.getCurrentThreadCpuTime();  
  
// some code  
  
long time2 = bean.getCurrentThreadCpuTime();
```

Sun's HotSpot JVM

- Sun's HotSpot JVM combines interpretation and Just-in-time (JIT) compilation
- Initially the bytecode is interpreted and JVM monitors which parts are executed frequently
- After reaching a compile threshold, “hot” methods and loops are compiled into native code
- JIT compiler uses the execution statistics collected by JVM during the interpretation to perform optimization

Sun's HotSpot JVM (cont.)

- Two modes in JDK 6
- HotSpot client, -client
 - default on Windows
 - focused on reduced startup time and footprint size
 - simple, quick optimizations
- HotSpot server, -server
 - focused on long running applications
 - sophisticated optimizations
- If not specified, ergonomics will choose automatically based on the hardware and software configuration

Hotspot JIT Compiler

You can change the default JIT compiler behavior using the following switches (don't use them except for experiments!):

- `-Xint` – don't compile, interpret only
- `-Xcomp` – compile everything, don't interpret

You can monitor the JIT compiler:

- `-XX:+PrintCompilation`
- `-XX:+LogCompilation`
- `CompilationMXBean`

-XX:+PrintCompilation

java -XX:+PrintCompilation ...

- 1 java.lang.String::charAt (33 bytes)
- 2 java.lang.String::hashCode (64 bytes)

[id][flags][class name::method name] [@ bci] ([size])

- id is ordinal number of compilation
- flags is none or more of:
 - % indicates “on stack replacement”
 - ! indicates that method has an exception handler
 - * indicates that method is native
 - b means that compilation is not done in parallel with execution
 - s means that method is synchronized
- bci is byte code index for an “on stack replacement” operation
- size is the size of the code generated

-XX:+LogCompilation

java -XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation ...

- the log file can be set by -XX:LogFile=... (default is hotspot.log)
- more detailed output than -XX:+PrintCompilation

```
<task compile_id='2' method='java/lang/String hashCode ()I' bytes='60'  
count='240' backedge_count='937' iicount='0' stamp='0.111'>  
  <task_done success='1' nmsize='176' count='240' backedge_count='937'  
    stamp='0.111'>  
</task>  
<task compile_id='3' method='java/lang/String indexOf (II)I' bytes='151'  
count='92' backedge_count='864' iicount='0' stamp='0.120'>  
  <task_done success='1' nmsize='492' count='92' backedge_count='864'  
    stamp='0.121'>  
</task>
```

CompilationMXBean

`getTotalCompilationTime()` returns the number of milliseconds that JIT compiler spent compiling the bytecode

```
CompilationMXBean bean = ManagementFactory.getCompilationMXBean();  
  
long time1 = bean.getTotalCompilationTime();  
  
// some code  
  
long time2 = bean.getTotalCompilationTime();
```

Demo: Compilation

1. Run the project in `jdk/demo/jfc/Java2D`
2. Run `jconsole` in `jdk/bin`
3. Inspect the VM summary tab
4. Inspect the `java.lang.Compilation` node in the Mbeans tab
5. Run the project again with the `PrintCompilation` flag

-XX:+PrintAssembly

- Must be preceded by -XX:+UnlockDiagnosticVMOptions
- Requires disassembler plugin
- More info on

<http://wikis.sun.com/display/HotSpotInternals/PrintAssembly>

Dynamic Optimization

- On stack replacement
- Dead-code elimination
- Escape analysis
- Autobox elision
- Inlining
- Loop unrolling

On Stack Replacement

JVM can replace the current code with new compiled code when executing the method

```
public static void main( String[] args ) {  
    int result = 0;  
    for ( int i = 0; i < 10000000; i++ ) {  
        for ( int j = 0; j < values.length; j++ ) {  
            result ^= values[j];  
        }  
    }  
    ...  
}
```

Dead-code Elimination

Dead code can be eliminated either by javac or by JVM

```
private static final boolean debug = false;

public static void main( String[] args ) {

    if ( debug ) {

        System.out.println( "start" );

    }

    ...

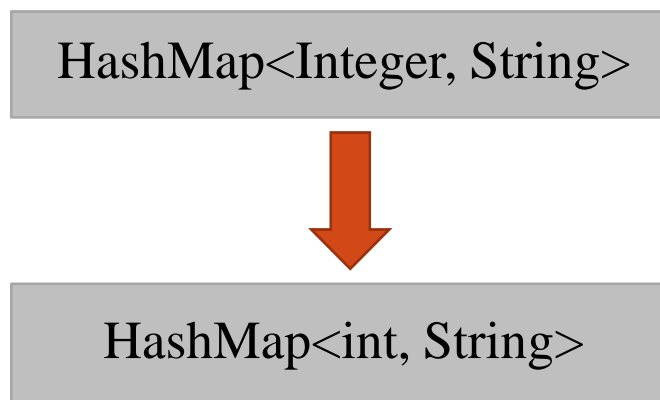
}
```


Escape Analysis

- An object *escapes* the thread that allocated it, if some other thread can see it
- Many objects are accessed by one thread only
- If an object does not escape, JVM can perform:
 - Thread stack allocation – fields are stored in the stack frame
 - Scalar replacement – scalar fields are stored in registers
 - Object explosion – object's fields are allocated in different places
 - Eliminate synchronization
 - Eliminate memory read / write barriers

Autobox Elision

- Autoboxing is conversion from primitive type to an instance of wrapper class
- Under some circumstances, JIT compiler can eliminate object allocation



Inlining

Inlining is an optimization that replaces a function call with the body of the function

```
static int max( int x, int y ) {  
  
    return x > y ? x : y;  
  
}
```

```
int m = max( i, j );
```



```
int m = i > j ? i : j;
```

Loop Unrolling

Loop unrolling is an optimization that optimize program's execution speed at the expense of its size

```
for ( int i = 0; i < 3000; i++ ) {  
    p[i] = 42;  
}
```



```
for ( int i = 0; i < 1000; i++ ) {  
    p[ i ] = 42;  
    p[ i + 1 ] = 42;  
    p[ i + 2 ] = 42;  
}
```

Micro-benchmark

- Measures performance of small piece of code
- JIT compiler can change what we measure (warm-up the JVM)
- Beware of the garbage collection
- `System.currentTimeMillis()` does not have millisecond accuracy
- `System.nanoTime()` does not have nanosecond accuracy
- Measure several times and interpret data statistically

Demo: Inlining

1. Open the project in demo/JavaVirtualMachine/Inlining in NetBeans
2. Get familiar with the source code
3. Run the project with the PrintCompilation flag
4. Run the project again with `-XX:MaxInlineSize=128`
5. Compare the results

Demo: Autoboxing Performance

1. Run the project in demo/JavaVirtualMachine/AutoboxingPerformance
2. Run the project again with `-XX:+PrintGC`
3. Run the project again with the following settings:
 - Initial heap size 1400 MB (`-Xms1400M`)
 - Maximum heap size 1400 MB (`-Xmx1400M`)
 - Young generation size 1000 MB (`-Xmn1000M`)
4. Run the project again with `-Djava.lang.Integer.IntegerCache.high=1000`
5. Compare the times

[From <http://www.javaspecialists.eu/archive/Issue191.html>]

Lab: Micro-benchmark

1. Open the project in labs/lab2/Microbenchmark in NetBeans
2. Get familiar with the source code (the micro-benchmark measures the times of iteration through List using Iterator and using `get(index)`)
3. Run the project and compare the times

Evaluation

```
3: aload_0
4: invokeinterface #11, 1; //InterfaceMethod java/util/List.iterator:()Ljava/util/Iterator;
9: astore_2
10: aload_2
11: invokeinterface #12, 1; //InterfaceMethod java/util/Iterator.hasNext:()Z
16: ifeq 45
19: aload_2
20: invokeinterface #13, 1; //InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
25: checkcast    #14; //class java/lang/Integer
28: astore_3
...
42: goto 10
```

Evaluation (cont.)

```
3:  iconst_0
4:  istore_2
5:  iload_2
6:  aload_0
7:  invokeinterface #16, 1; //InterfaceMethod java/util/List.size:()I
12: if_icmpge      45
15:  aload_0
16:  iload_2
17:  invokeinterface #17, 2; //InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
22:  checkcast     #14; //class java/lang/Integer
25:  astore_3
...
39:  iinc    2, 1
42:  goto    5
```

Lab: Lazy Bezier

1. Run the project in labs/lab2/LazyBezier
2. Monitor the project in Visual VM
3. Find out what is keeping threads out of CPU
4. Propose code changes so that CPU is fully utilized

Lab: Lazy Textures

1. Run the project in labs/lab2/LazyTextures
2. Profile the project
3. Find the hot spot

Conclusion

- Java Virtual Machine (JVM)
- Java disassembler
- Just-in-Time (JIT) compiler
- Micro-benchmark

Questions & Answers

Next module: Memory management

3 Memory Management

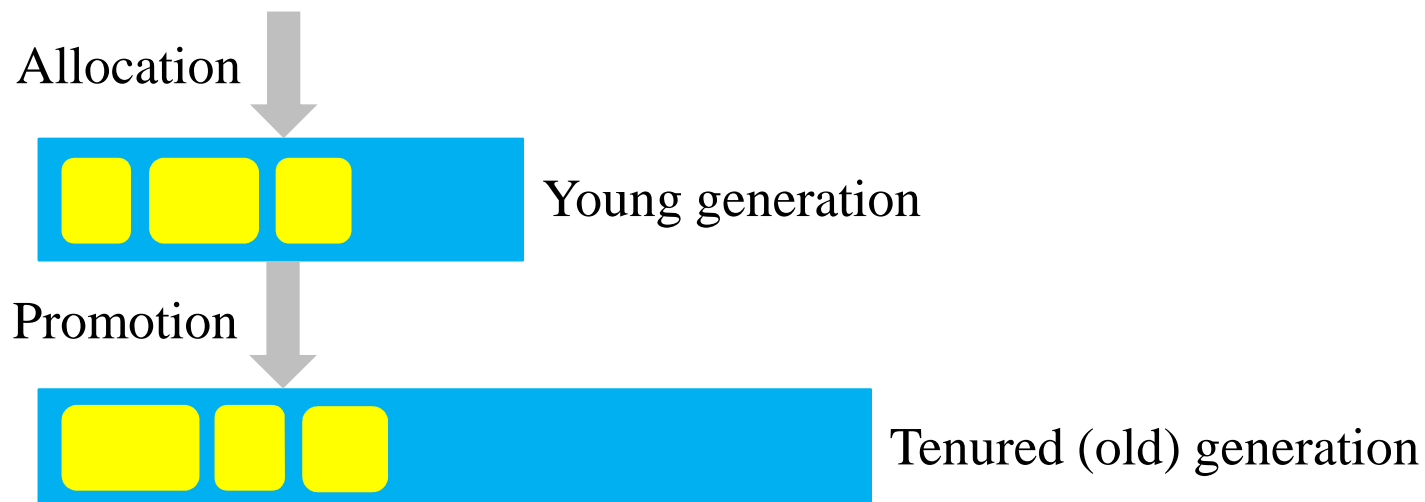
Module Content

- Java heap
- Garbage collectors
- Monitoring garbage collector
- Memory management tuning

Oracle JVM

Weak Generational Hypothesis

- Most objects die young
- Few references from older to younger objects exist



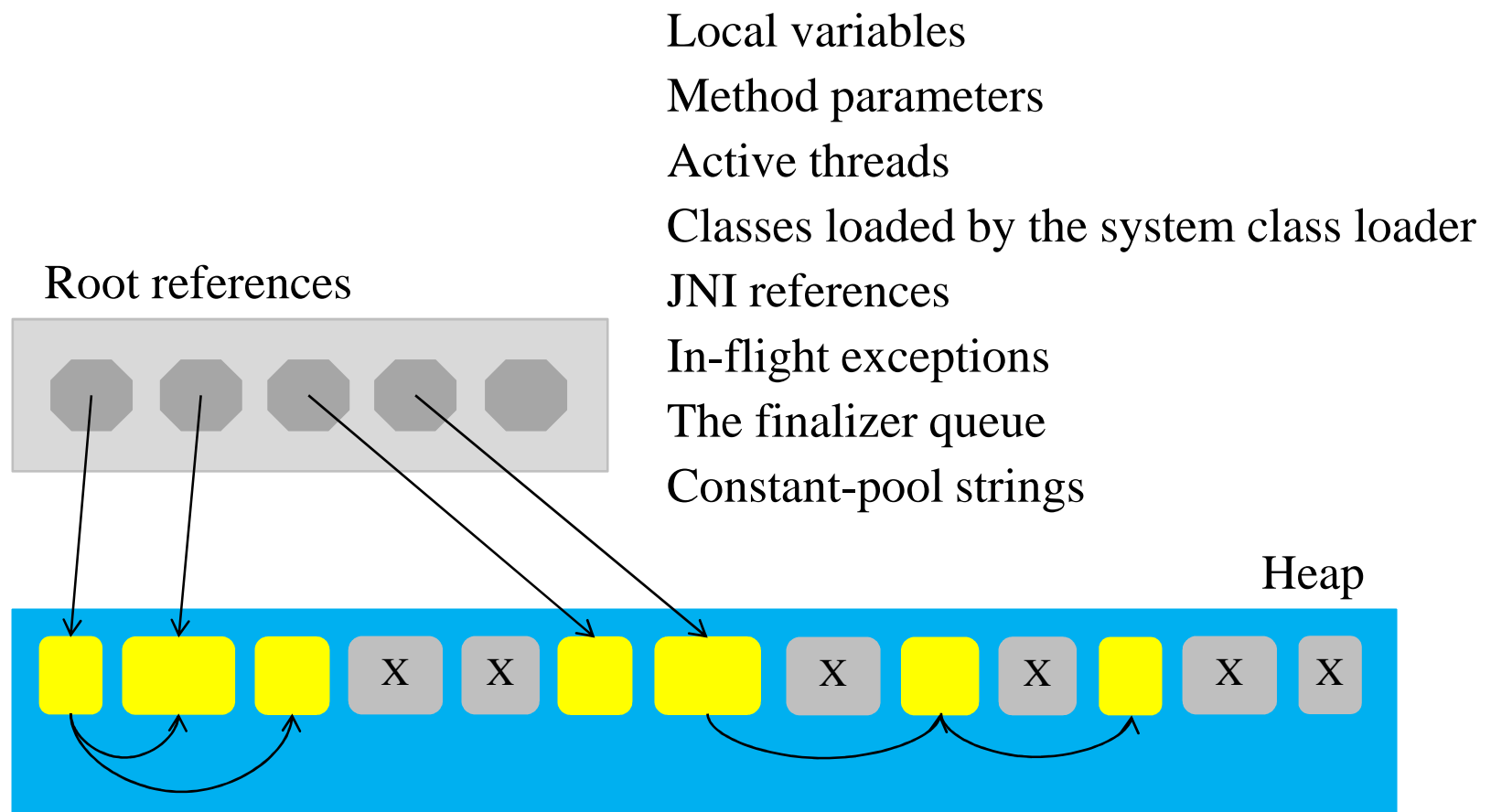
Generational Algorithm

- Young generation – “young” objects
- Tenured (old) generation – “old” or large objects
- Permanent generation – “permanent” objects, e.g. instances of Class, Method, and Field

Garbage Collections

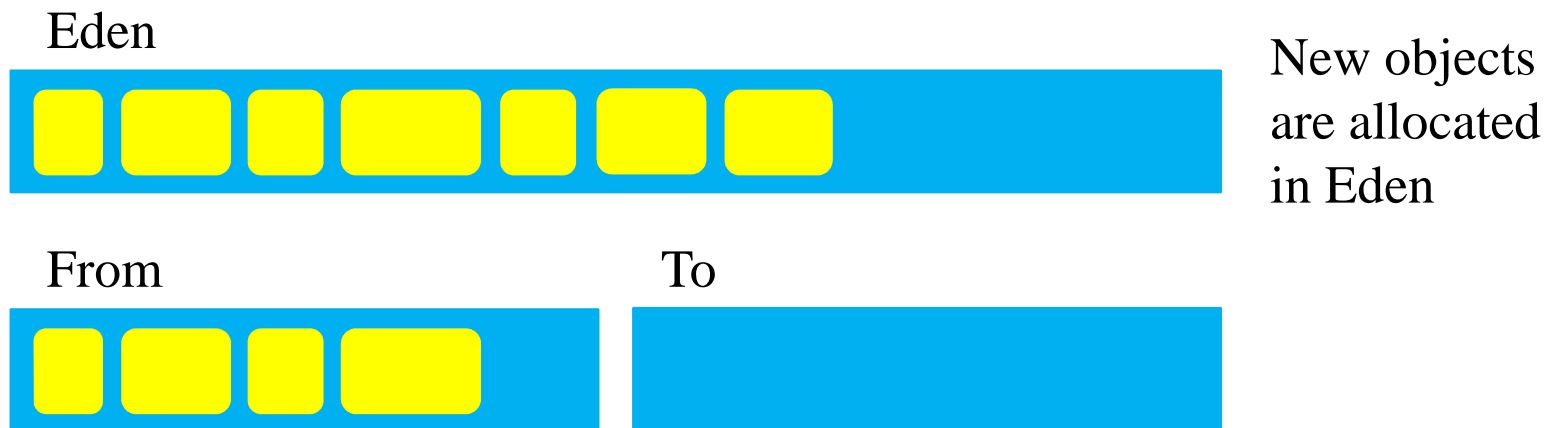
- “Stop the world” – all Java application threads are blocked for the duration of the garbage collection
- Minor collection: young generation
- Major (full) collection: all heap
- Usually different algorithms are used for minor and major collections

Mark & Sweep

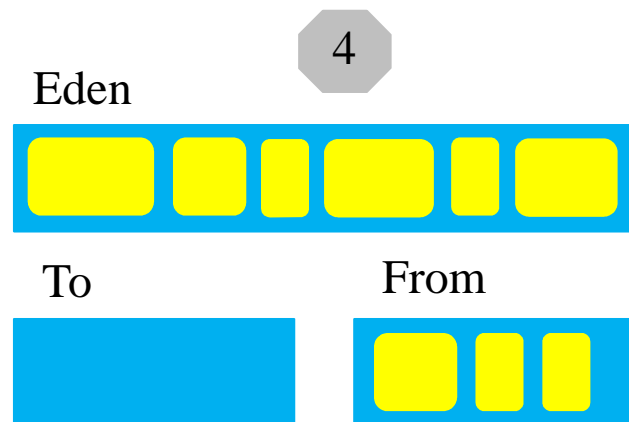
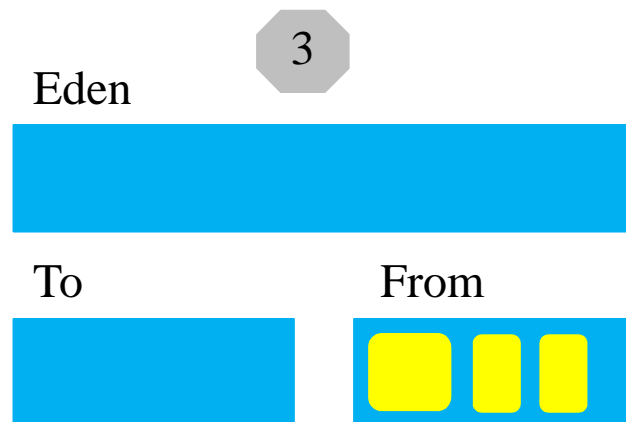
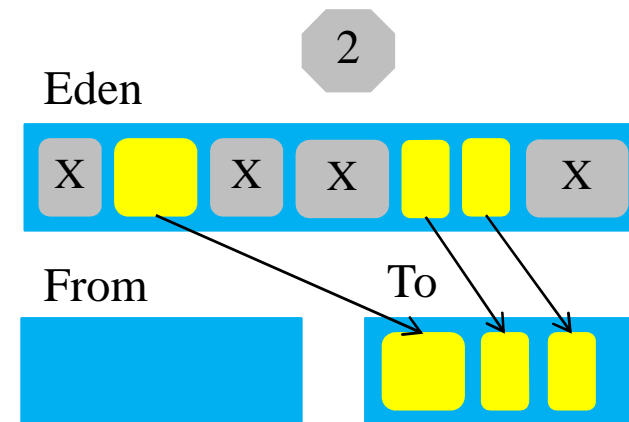
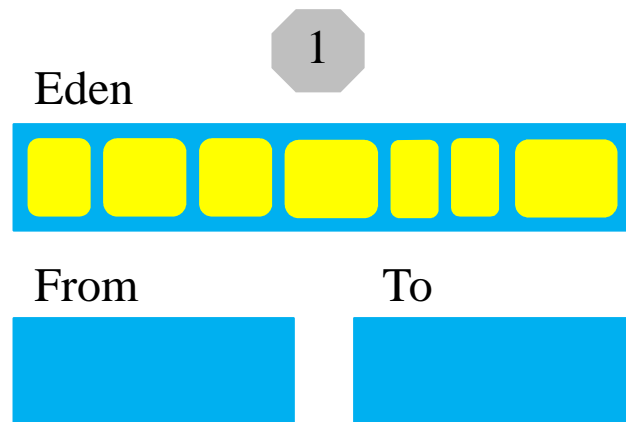


Young Generation

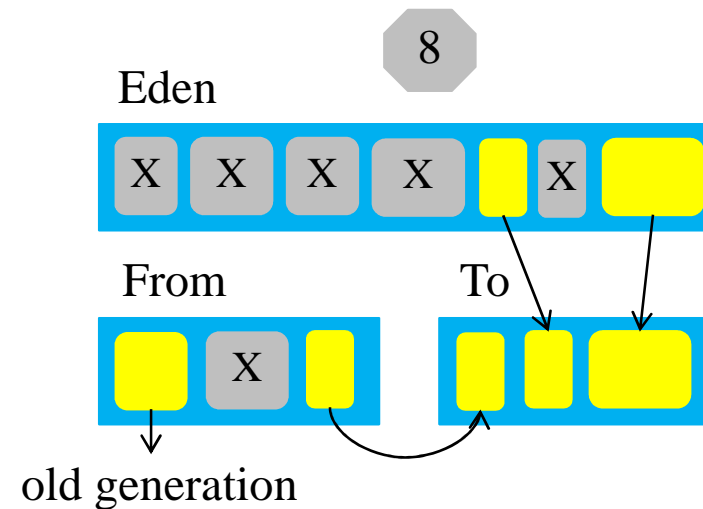
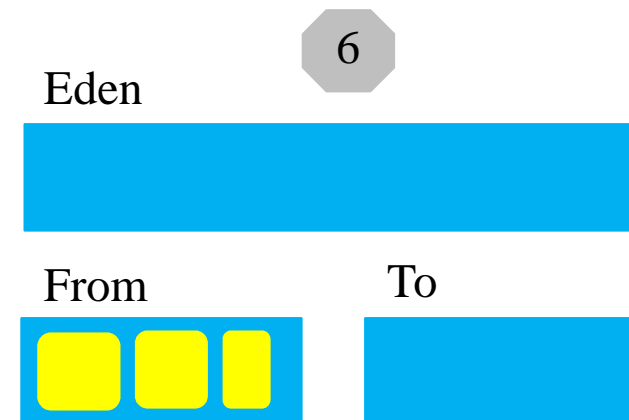
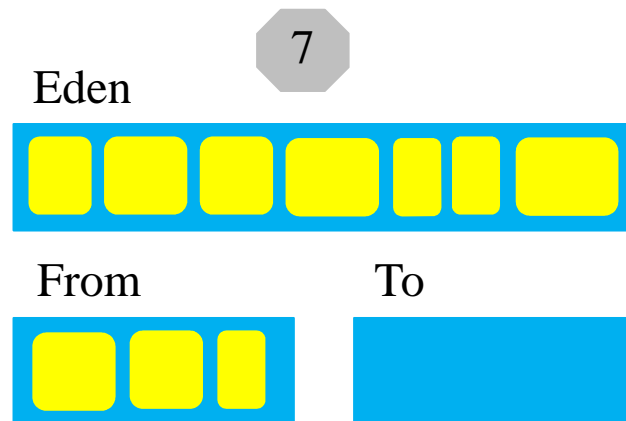
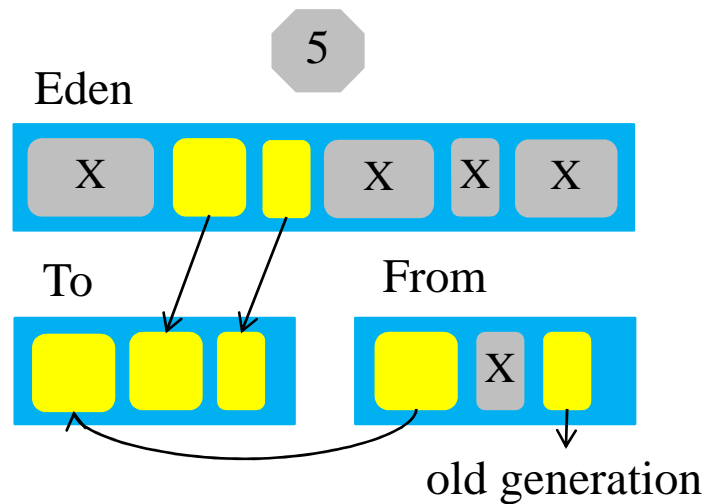
- Three areas: Eden and two Survivor spaces (From and To)
- Copying collector



Minor Collection



Minor Collection (cont.)



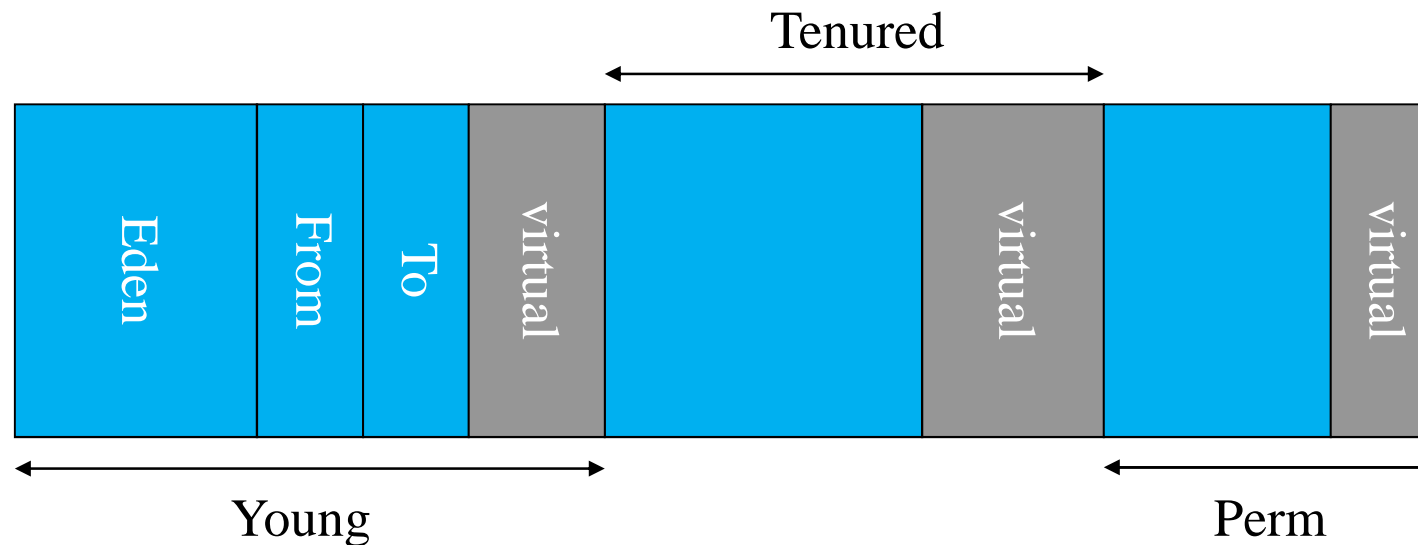
Minor Collection (cont.)

- When the Eden space is full, minor garbage collection event occurs
- Each object that survives a garbage collection has its age incremented
- Objects exceeding a defined age threshold are promoted to the tenured generation space

Minor Collection (cont.)

- When there is not enough room in survivor spaces, the full garbage collection event occurs
- Minor garbage collections are very quick compared to full garbage collections

Sizing Java Heap



- -Xmx – maximum size of Java heap (young generation + tenured generation)
- -Xms – initial size of Java heap (young generation + tenured generation)
- -Xmn – size of young generation

Sizing Java Heap (cont.)

- `-XX:NewSize` – initial size of young generation
- `-XX:MaxNewSize` – maximum size of young generation
- `-XX:NewRatio` – ratio of young generation to tenured generation
- `-XX:PermSize` – initial size of permanent generation
- `-XX:MaxPermSize` – maximum size of permanent generation

Sizing Java Heap (cont.)

For application performance we almost always set

- `-Xms` and `-Xmx`
- `-XX:PermSize` and `-XX:MaxPermSize`

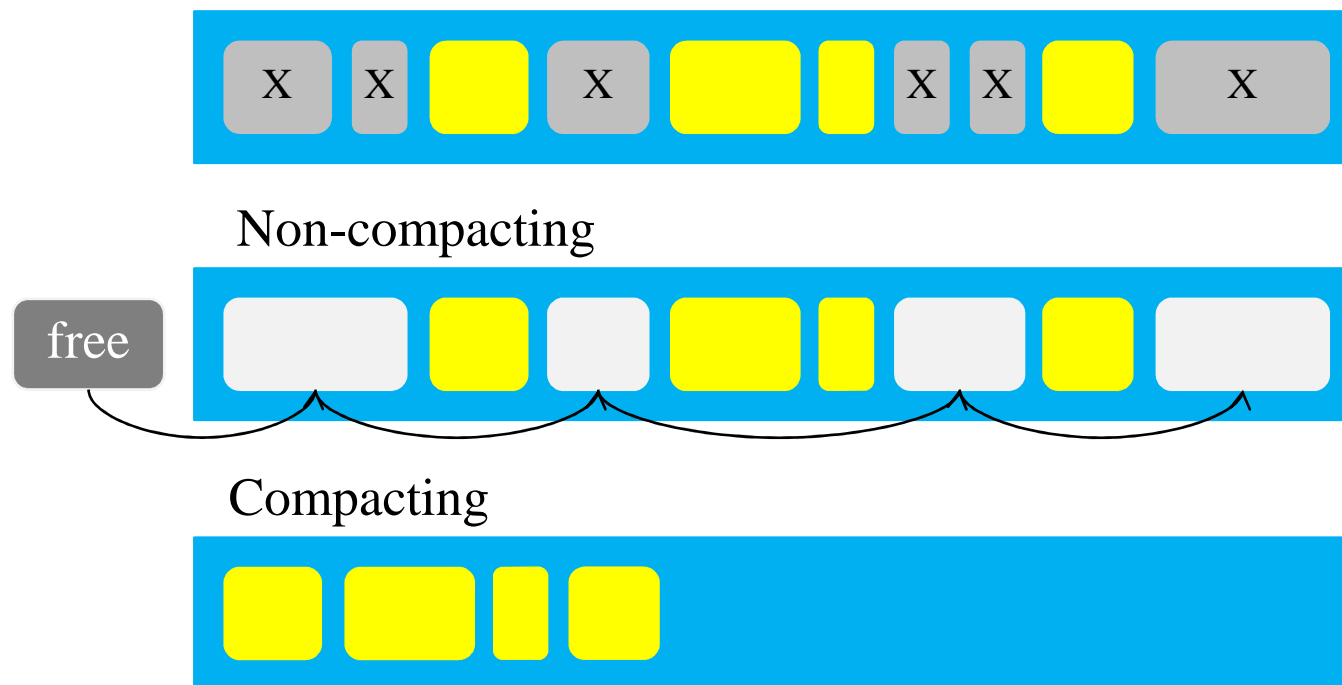
to the same values because growing the space requires full garbage collection

Demo: Visual GC

1. Run the project in `jdk/demo/jfc/Java2D` with
`-XX:+DisableExplicitGC`
2. Run Visual VM in `jdk/bin`
3. Open the project in Visual VM and switch to the Visual GC tab
4. Observe a saw tooth pattern on the Eden space

GC Features

- Serial vs. Parallel
- Stop-the-world vs. Concurrent
- Copying vs. Compacting vs. Non-compacting



GC Performance Metrics

- Throughput – the percentage of total time not spent in garbage collection, considered over long periods of time
- GC overhead – the percentage of total time spent in garbage collection
- Pause time – the length of time when application is stopped due to garbage collection

GC Performance Metrics (cont.)

- Frequency of collection – how often collection occurs within some time period
- Footprint – how much memory the garbage collector needs
- Promptness – the time between when an object becomes garbage and when it is collected

Garbage Collectors

- Serial collector
- Throughput collector
- Concurrent collector

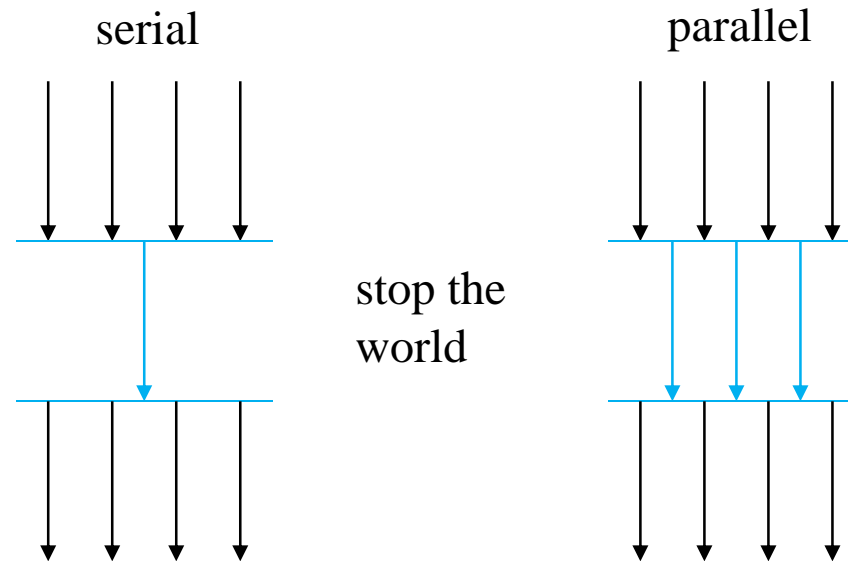
Serial Collector

- Single threaded collector
- Selected by `-XX:+UseSerialGC` (default on client machines)
- Young generation: copy collector
- Tenured generation: mark-sweep-compact
- Suitable for single processor core machine

Throughput (Parallel) Collector

- Selected by `-XX:+UseParallelGC` (default on server machines)
- Young generation: multi-threaded copy collector
- Tenured generation: single-threaded mark-sweep-compact
- Suitable for multi-core processor systems

Serial vs. Parallel Collector



Demo: Memory Consumer

1. Run the project in demo/MemoryManagement/MemoryConsumer
with the following settings:
 - Initial and maximum heap size 1024 MB
 - Serial garbage collector
2. Run Visual VM and open the project
3. Observe the Visual GC tab

Demo: Memory Consumer (cont.)

1. Run the project in demo/MemoryManagement/MemoryConsumer
with the following settings:
 - Initial and maximum heap size 1024 MB
 - Parallel garbage collector
2. Open the project in Visual VM
3. Observe the Visual GC tab

Parallel Compacting Collector

- Selected by `-XX:+UseParallelOldGC`
- Young generation: multi-threaded copy collector
- Tenured generation: multi-threaded mark-sweep-compact
- Suitable for multi-core processor systems
- The number of threads can be controlled with `-XX:ParallelGCThreads`

Default settings

$\text{ParallelGCThreads} = (\text{nps} \leq 8) ? \text{nps} : 3 + ((\text{nps} * 5) / 8)$

where nps is the number of processors

Concurrent Collector (CMS)

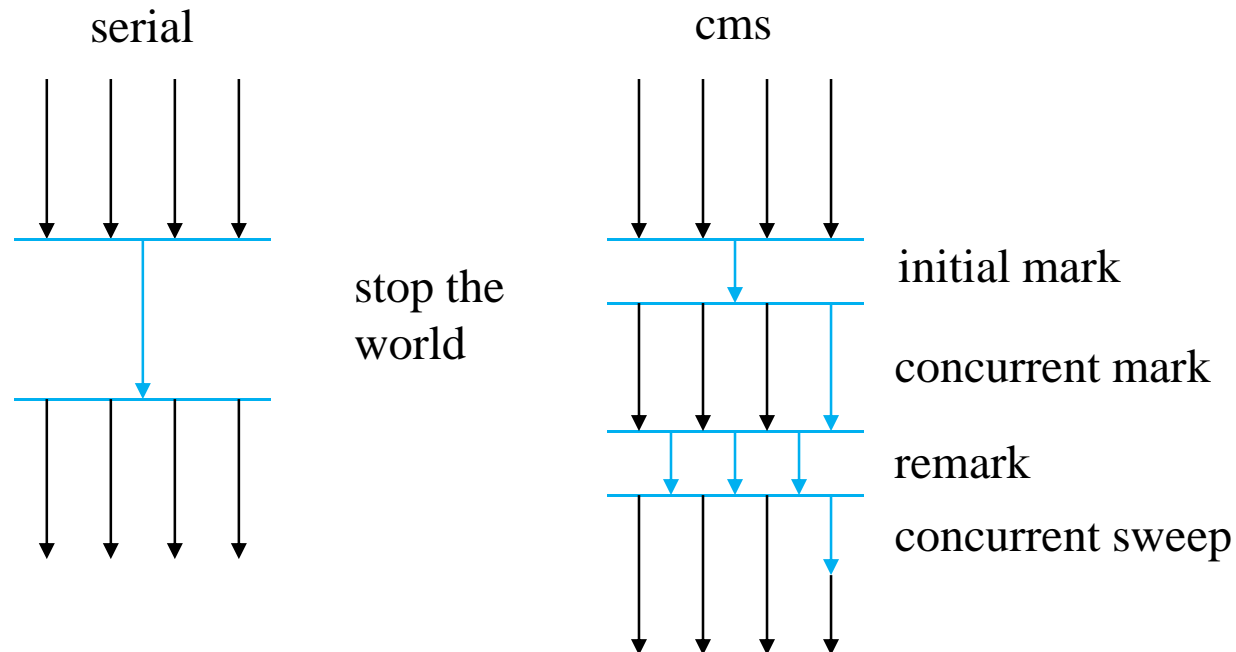
- Selected by `-XX:UseConcMarkSweepGC`
- Multi-threaded young generation collector (same as in throughput collector)
- Single threaded tenure generation collector that runs concurrently with Java application threads (most of the work is done concurrently with application)
- Does not compact memory
- Suitable when application responsiveness is more important than application throughput

Concurrent Collector Phases

- Initial mark phase: marks objects directly reachable from application
- Concurrent mark phase: traverses the object graph concurrently with Java application threads
- Remark: finds objects that were missed by the concurrent mark phase due to updates by Java application threads
- Concurrent sweep: collects the objects identified as unreachable during marking phases
- Concurrent reset: prepares for next concurrent collection

Serial vs. CMS Collector

major collection:



CMS Incremental Mode

- Selected by `-XX:+CMSIncrementalMode`
- Concurrent phases are done incrementally
- Collector periodically gives processor back to the application

Explicit GC

- Explicit GC can be disabled with `-XX:+DisableExplicitGC`
- Do not use `System.gc()` unless you have a strong reason (e.g. in performance tests)
- `-XX:+DisableExplicitGC` disables RMI distributed garbage collection

Lab: Serial & Parallel GC

1. Run the project in `jdk/demo/jfc/Java2D` with the following settings:
 - Initial and maximum heap size 32 MB
 - Young generation size 16 MB
 - Initial and maximum permanent generation size 16 MB
 - Explicit GC disabled
 - Serial or parallel GC
2. Measure time for 10 minor collections
3. Compare performance of the serial and parallel GCs

Finalizers

- Finalizers are not destructors
- Object cannot be collected until the finalizer is executed
- Do not use finalizers to manage resources other than memory
- Finalizers can be a safety net (but try to limit their use)
- If you use finalizers, keep the work being done in them as small as possible

Ergonomics

- Evaluates the system and chooses defaults for the HotSpot JVM
- Uses definition of “server class machine”:
 - 2 or more processor cores and 2 or more GB of physical memory
 - special case: 32-bit Windows machine are never server class
 - special case: 64-bit machine are always server class
- If a machine is considered server class, the JVM runs in server mode
- Otherwise, the JVM runs in client mode

Server Mode

- Server JIT compiler
- Throughput collector
- Initial heap size is $1/64^{\text{th}}$ of the physical memory up to 1 GB (i.e. the minimum initial heap size is 32 MB)
- Maximum heap size is $1/4^{\text{th}}$ of the physical memory up to 1 GB

Client Mode

- Client JIT compiler
- Serial garbage collector
- Initial heap size is 4 MB
- Maximum heap size is 64 MB

Parallel Collector Tuning

- `-XX:MaxGCPauseMillis= n` – specifies maximum desired pause time
- `-XX:GCTimeRatio= n` – specifies desired throughput

The ratio of garbage collection time to application time is

$$1 / (1 + n)$$

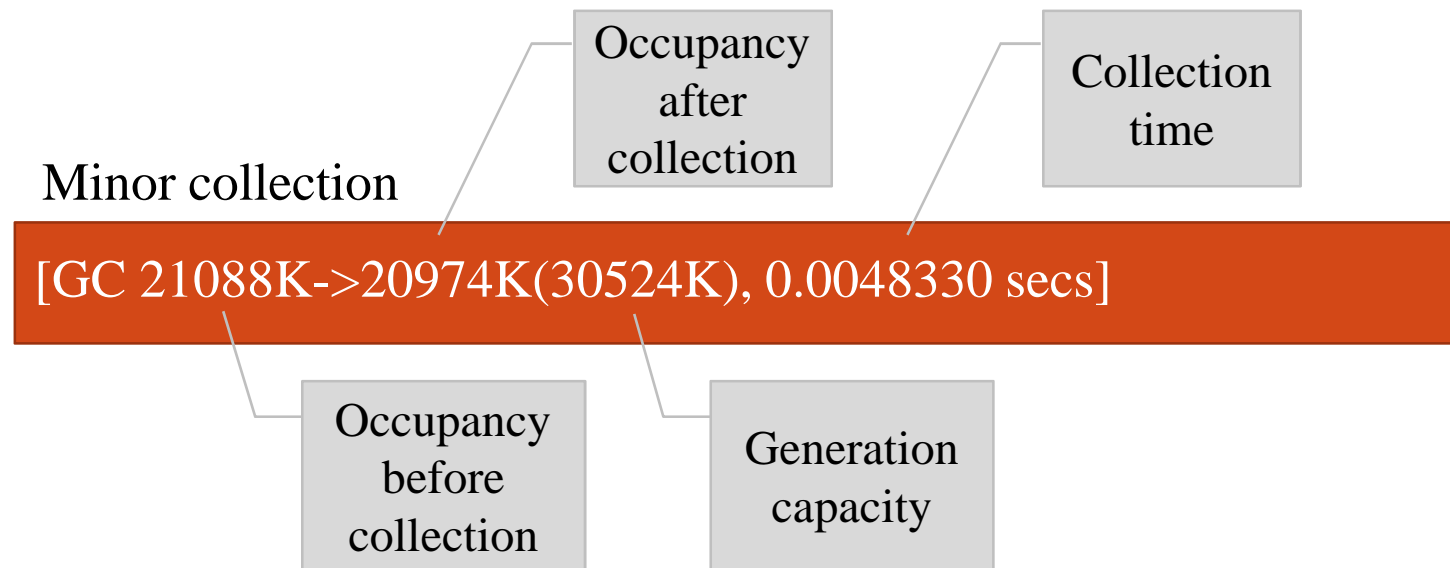
Example: $n=19$ means a goal of 5 % of the total time for GC

JVM will automatically and dynamically modify the heap sizes
in order to achieve the desired goals

GC Monitoring

- `-verbose:gc`, `-XX:+PrintGC`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`, `-XX:+PrintGCDateStamps`
- `-XX:+PrintHeapAtGC`
- `-XX:+PrintGCApplicationStoppedTime`
- Visual VM, Visual GC
- jconsole

-verbose:gc, +XX:+PrintGC



Major collection

[Full GC 29089K->29089K(37692K), 0.0061875 secs]

-XX:+PrintGCDetails

Minor collection

```
[GC [DefNew: 4415K->512K(4928K), 0.0036650 secs]
4415K->4285K(15872K), 0.0038643 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```

Young
generation



```
[GC [<collector>: <starting occupancy1> -> <ending occupancy1>
(generation size), <pause time1> secs] <starting occupancy3> ->
<ending occupancy3>(heap size), <pause time3> secs]
```



Entire heap

-XX:+PrintGCDetails (cont.)

Minor collection with promotion

```
[GC [DefNew: 4906K->503K(4928K), 0.0031987 secs]  
[Tenured: 12565K->12591K(12608K), 0.0066095 secs]  
13080K->13068K(17536K), [Perm : 44K->44K(12288K)],  
0.0102299 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

Major collection

```
[Full GC [Tenured: 174774K->174774K(174784K), 0.0226224 secs]  
253429K->252856K(253440K), [Perm : 43K->43K(12288K)],  
0.0228810 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
```

-XX:+PrintGCTimeStamps

```
java -XX:+PrintGC -XX:+PrintGCTimeStamps ...
```

Time when
collection started

Duration of
collection

11.875: [GC 12178K->7777K(17588K), 0.0039663 secs]

15.855: [Full GC 8943K->8643K(21312K), 0.0620489 secs]

-XX:+PrintGCApplicationStoppedTime

```
java -XX:+PrintGC -XX:+PrintGCTimeStamps  
-XX:+PrintGCApplicationStoppedTime ...
```

```
12.777: [Full GC 8217K->6877K(16620K), 0.0466502 secs]  
Total time for which application threads were stopped: 0.0477718 seconds  
13.456: [Full GC 7569K->7280K(16620K), 0.0520881 secs]  
Total time for which application threads were stopped: 0.0532108 seconds  
Total time for which application threads were stopped: 0.0001046 seconds  
Total time for which application threads were stopped: 0.0000757 seconds
```

Application was
stopped but no
collection started

Common Problems

- Young generation too small
- Old generation too small
- Perm generation too small

Young Generation Too Small

- Young generation is filled up before temporary objects get unreachable
- Temporary objects do not die in young generation and are promoted to tenured generation
- Consequences: garbage collection takes more time due to excessive copying, major collections are frequent

Young Generation Too Small (cont.)

Young generation: 4096 KB

Eden: 3328 KB

Survivor 0: 384 KB

Survivor 1: 384 KB

Capacity
of young
generation

[GC [DefNew: 3328K->384K(3712K), 0.0045470 secs]
5376K->4416K(32384K), 0.0047182 secs]

Change of occupancy in young generation: $3328 - 384 = 2944$ KB

Change of occupancy over entire heap: $5376 - 4416 = 960$ KB

Size of promoted objects: $2944 - 960 = 1984$ KB

Ratio of promoted objects: $1984 / 2944 = 67 \%$

Old Generation Too Small

- Manifestation: major collections are frequent and occur at the same time as minor collections
- Before a minor collection starts, garbage collector checks, if there is enough room for objects that will probably be promoted to the old generation
- If there is not enough room, a major collection happens

Demo: Old Generation Too Small

- Run the project in `jdk/demo/jfc/Java2D` with the following settings:
 - Initial and maximum heap size 12 MB
 - Initial and maximum size of young generation 4 MB
 - Verbose GC
 - Explicit GC disabled
- Observe the output

Perm Generation Too Small

- You get an exception:

`java.lang.OutOfMemoryError: PermGen space`

- Solution: increase the size of permanent generation

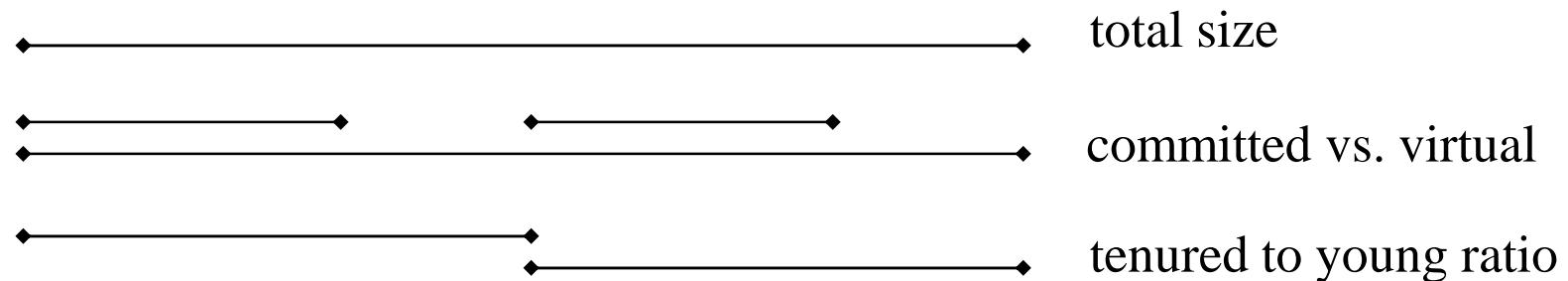
Demo: Perm Generation Too Small

- Run the project in `jdk/demo/jfc/Java2D` with the following settings:
 - Initial and maximum size of permanent generation 3 MB
- Use the application until `OutOfMemoryError` appears

GC Tuning

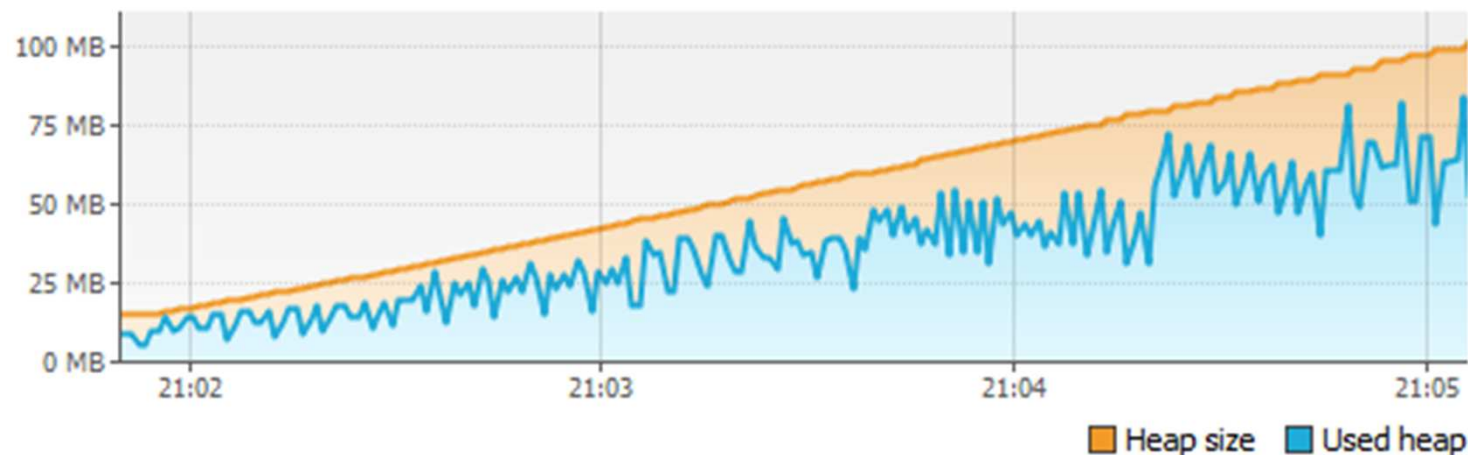
- Are minor collection pauses too long? Try parallel collector.
- Are major collection pauses too long? Try concurrent collector.

Heap Size Settings



Memory Leak

Also known as unintentional object retention (the object that is not needed anymore and should be unreachable is still reachable and cannot be collected)



Basic Types of Memory Leaks

- A single large object that is unintentionally held in memory
- Small objects that are accumulated over time (e.g. they are being added to a collection)









Finding Memory Leaks

- Generations in memory profiler
- Heap snapshots comparison
- Heap dumps

Generations in Memory Profiler

Object age – the number of garbage collections that the object survived

Generations – the number of different object ages for objects of given class (counted over all alive objects on the heap)

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Generati...
memoryleak.Result		720 B (1,9%)	45 (6,8%)	45
java.lang.ref.WeakReference		552 B (1,5%)	23 (3,5%)	23
char[]		4 448 B (11,9%)	68 (10,2%)	3
java.lang.String		864 B (2,3%)	36 (5,4%)	3
byte[]		11 46... (30,7%)	10 (1,5%)	2
java.lang.Object[]		2 744 B (7,4%)	61 (9,2%)	2
java.io.ObjectStreamClass\$WeakClassKey		1 504 B (4%)	47 (7,1%)	2
java.util.TreeMap\$Entry		1 280 B (2,4%)	40 (6,8%)	2

[Class Name Filter]

Demo: Memory Leak

1. Run the project in demo/MemoryManagement/MemoryLeak
2. Run Visual VM and open the project
3. Monitor the project

Heap Snapshots Comparison

- Take memory snapshots in different points of application (it is advisable to run garbage collector before)
- Compare the snapshots (e.g. in NetBeans or in Visual VM)
- Search for leaking objects within the objects that were identified as new

Demo: Lucky Numbers

1. Open the project in demo/MemoryManagement/LuckyNumbers in NetBeans
2. Profile the project in NetBeans
3. Use memory snapshots comparison to identify the memory leak

Demo: Stop-time Game

1. Open the project in demo/MemoryManagement/StopTimeGame in NetBeans
2. Profile the project in NetBeans
3. Find and explain the memory leak

Lab: Nasty Ellipses

1. Open the project in labs/lab3/NastyEllipses in NetBeans
2. Run the project
3. Check if there is a memory leak and if yes find the cause

Lab: Slow Baker

1. Open the project in labs/lab3/SlowBaker in NetBeans
2. Run the project
3. Check if there is a memory leak and if yes find the cause

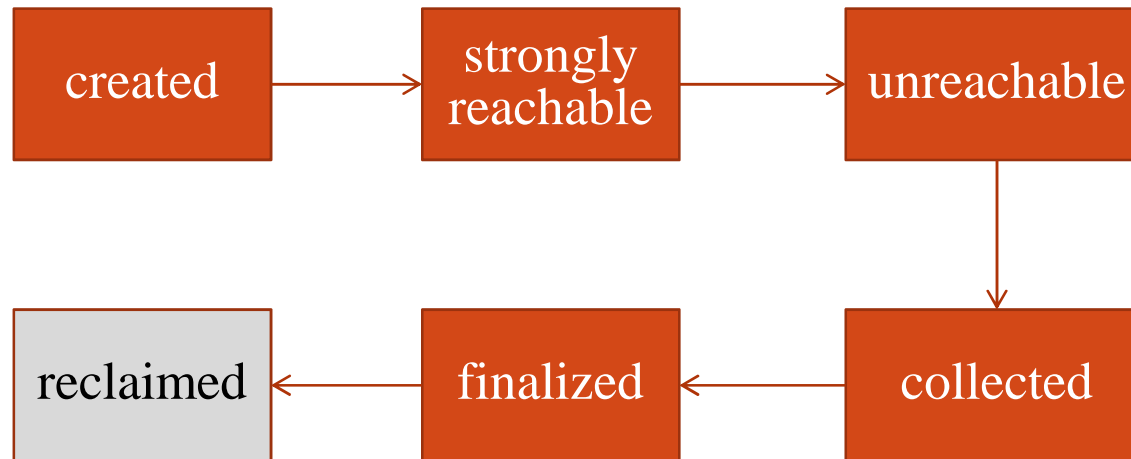
References (java.lang.ref)

- Strong reference – a common Java reference
`Car c = new Car(id);`
- Weak reference – a reference that does not prevent garbage collector to collect the object which the reference refers to
`WeakReference<Car> p = new WeakReference<Car>(c);`
- Soft reference – like a weak reference, except that the garbage collector is less eager to collect the object which the reference refers to
`SoftReference<Car> p = new SoftReference<Car>(c);`
- Phantom reference – a reference that is enqueued after the garbage collector determines that the object which the reference refers to may be reclaimed
`ReferenceQueue<Car> queue = new ReferenceQueue<Car>();`
`PhantomReference<Car> p = new PhantomReference<Car>(c, queue);`

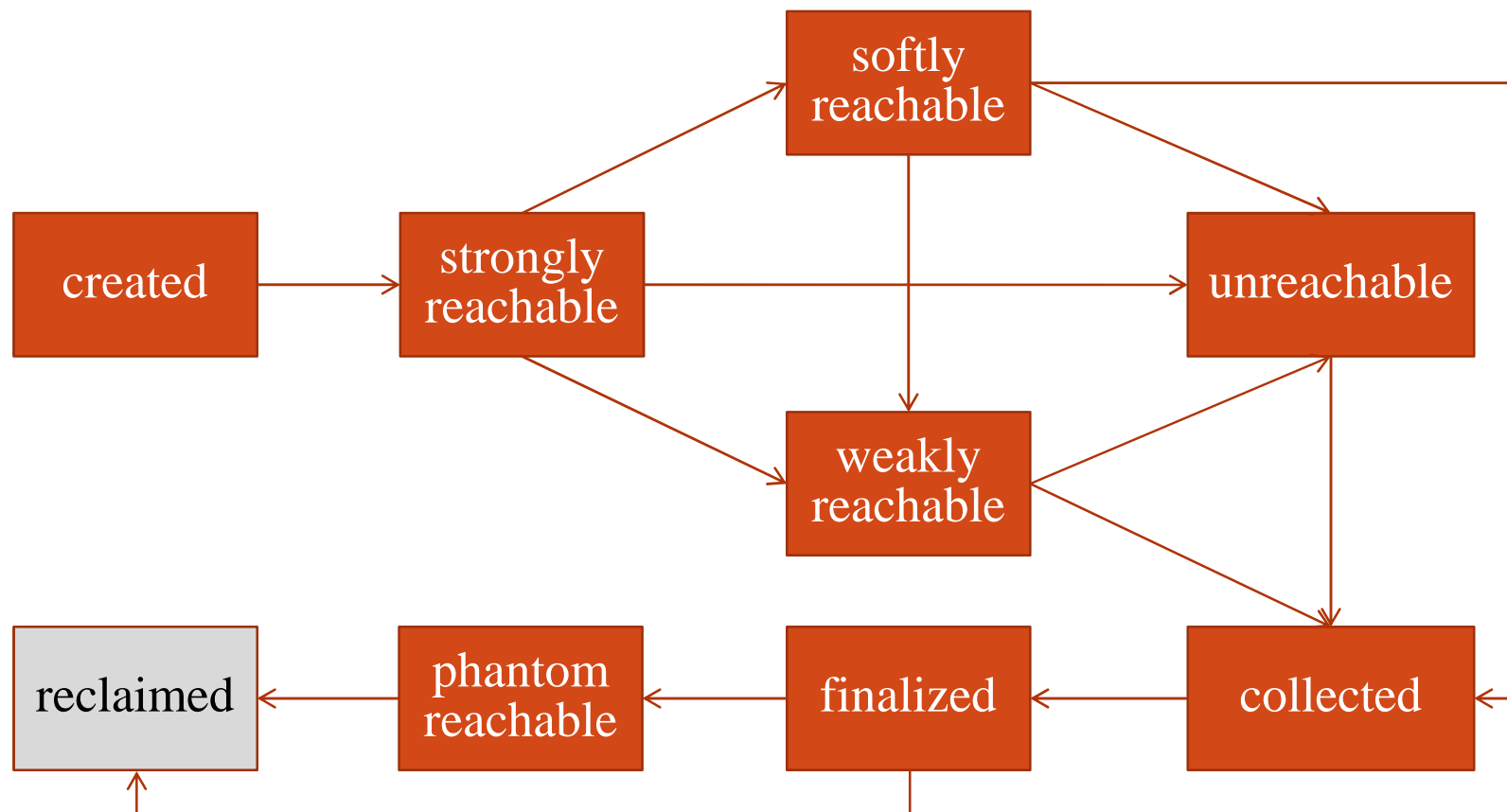
Object Reachability

- Strong reachability – an object is strongly reachable if it can be reached without traversing any reference object
- Soft reachability – an object is softly reachable if it is not strongly reachable and can be reached by traversing a soft reference
- Weak reachability – an object is weakly reachable if it is neither strongly nor softly reachable and can be reached by traversing a weak reference
- Phantom reachability – an object is phantom reachable if it is neither strongly, softly, nor weakly reachable, it has been finalized and some phantom reference refers to it

Object Reachability (cont.)



Object Reachability (cont.)



Demo: References

1. Open the project in demo/MemoryManagement/References in NetBeans
2. Run the project
3. Use jmap to create heap dump
4. Use VisualVM to run garbage collector
5. Create another heap dump
6. Open heap dumps in VisualVM and compare them

String

- Immutable sequence of characters
- Compiler places string literals into the constant pool
- Strings in constant pool are never garbage collected
- `String.intern()` returns the canonical representation
- Interned strings are garbage collected

WeakHashMap

- A hash-table based Map implementation with weak keys
- When a key is no longer in use, the entry is automatically removed by the garbage collector

Demo: WeakHashMap

1. Open the project in demo/MemoryManagement/WeakHashMap in NetBeans
2. Open Demo1 and determine the output
3. Run Demo1 and verify your answer
4. Open Demo2 and determine the output
5. Run Demo2 and verify your answer

Lab: Web Browser

1. Open the project in labs/lab3/WebBrowser in NetBeans
2. Run the project
3. Find the memory leak and its cause

OutOfMemoryError

Additional message:

- Java heap space
- PermGen space
- Requested array size exceeds VM limit

Heap Dump

We can dump all reachable objects to a file by

- Visual VM
- jmap
- `-XX:+HeapDumpOnOutOfMemoryError`

The file can be eventually analyzed

Lab: Hungry Duke

1. Run HungryDuke.jar with the following settings:
 - Maximum heap size 32 MB
 - Flag HeapDumpOnOutOfMemoryError
2. Wait for OutOfMemoryError
3. Analyze the heap dump in Visual VM
4. Find the cause of memory leak

Questions & Answers

Next module: Threads and Synchronization