

Extreme Java



Rony keren

Topics 1

- JVM
 - Roles
 - Modes
 - Features
 - Memory types
 - Weak References
- GC
 - Introduction
 - GC types
 - GC Algorithms
 - Heap generations
 - G1
 - Tuning
- Class-Loading
 - Introduction
 - Endorsed, Ext and Classpath
 - Build-in mechanism
 - Custom class loading
 - Multithreaded CL
 - Java Mission Control (JMC)
- Java Concurrency
 - Basics brief
 - Executors
 - Callable & Future
 - Atomic Concurrent
 - Lock API (Lock, Criteria)
 - Barriers
 - Fork-Join
 - Concurrent Collections
 - Adders

Topics 2

- Reflection
 - The need
 - Loading classes
 - Exploring class constructors, methods and fields
 - Manipulate objects with reflection
 - Handling arrays
 - Handling annotations
 - Reflecting parameter names (Java8)
- NIO, NIO.2
 - NIO buffers
 - Encoders & decoders
 - NIO.2 file system API – intro
 - NIO.2 asynchronous channels
 - NIO.2 watch services – FS change detection
- Functional Programming & Streams API
 - Understanding invoke dynamic
 - @FunctionalInterface
 - Predicate, Function, Consumer, Supplier
 - Default methods for interfaces
 - Using LAMBDA expressions
 - Using method references
 - Functional programming with Stream API
 - Intro to parallel streams
 - Common pool and dedicated pools



JVM Internals



- Roles
- Modes
- Memory types
- Weak references

Virtual Machine

VM Data areas:

- Heap
 - for Objects and Arrays
 - shared by all VM threads
- Stack
 - command and block sequence
 - recursive
 - Thread scoped – each thread has its own stack
- Metaspace— Permanent
 - loaded classes
 - constant string pool

Virtual Machine

VM Data areas:

- Java 6
 - Classes and interned strings were allocated in the PermGen which was part of the heap
- Java 7
 - Puts interned strings in a new place – Metaspace
- Java 8 doesn't come with a PermGen at all. Only Metaspace.
 - Classes are also allocated on Metaspace
- Metaspase
 - Native space used for holding classes and interned strings
 - Faster than PermGen
 - By default – has no max size –
 - Allocates memory according to needs (unlike PermGen which was part of the heap)
 - Native memory reduces footprint and executes faster

Virtual Machine

Byte code execution

- JDK 1.0 - Direct interpretation
 - Simple VM implementation – small footprint
 - Problem: Slow. No-cache
- JDK 1.2 – JIT
 - Caching of byte code for re-use
 - Problem: most optimizations are for client side code
- JDK 1.3-4 – Hotspot
 - Tracks hotspots for both server & client applications
 - VM uses a global information for optimization (like peek blocks/flows)
 - Problem: Heavy VM, big footprint

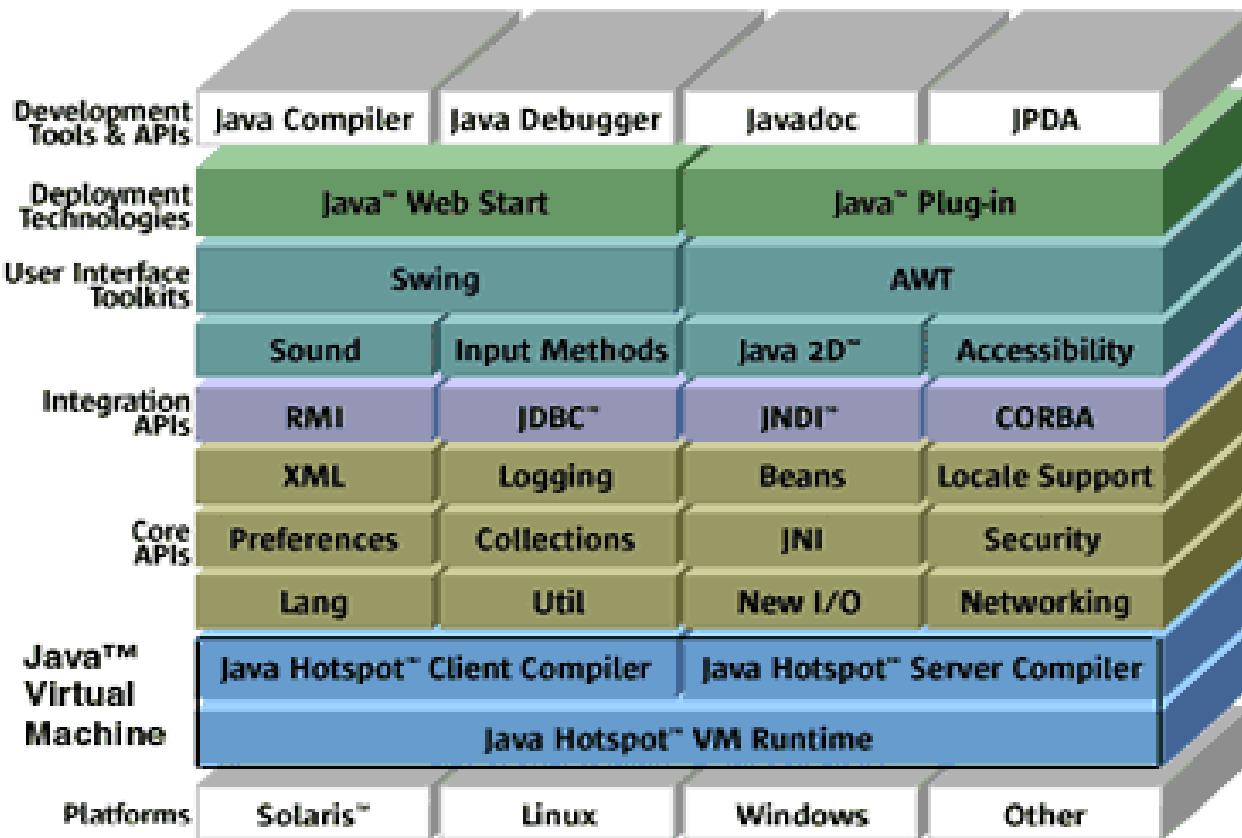
Byte code execution

- JDK 5.0 - Byte-code cache and share
- JDK 6.0 - API for scripting engines
- JDK 7.0 – Dynamic invocation for scripts
- JDK 8.0 – LAMBDA exp. Support
 - Smaller JVM footprint – heading toward embedded Java
 - Streaming API – getting ready for clouds and big-data

Hotspot

Enhanced JVM

Java™ 2 Platform, Standard Edition v 1.4



JVM Modes:

- Client mode – compiles instant used code - java –client ...
 - Assuming that code will be reused frequently
 - Compilation starts sooner than in server mode
 - Fast launching – available JIT compilations for immediate execution
- Server mode – compiles after more runtime analysis - java –server ...
 - Assuming there is much going on so better sit and analyze...
 - Optimizations are much more sophisticated
 - Slow launching – analyzing..... Code is re-interpreted until JIT kicks in...
- Tiered compilation
 - Java 7 – experimental
 - Java 8 - default

JVM Modes:

- Client mode defaults
 - JIT compilation after 1500 invocations
- Server mode defaults
 - JIT compilation after 10000 invocations
- Manually set via : -XX:CompileThreshold=<int value>

JVM Modes:

- Java 8 - Tiered Compilation by default
 - Starts in client mode for fast launching and then shifts into server mode
 - Was experimental in Java 7 (XX:+TieredCompilation)
 - Default in Java 8 (for –server supported environments)
 - Code fragments are being discarded or refactored along the way

Virtual Machine

Memory management

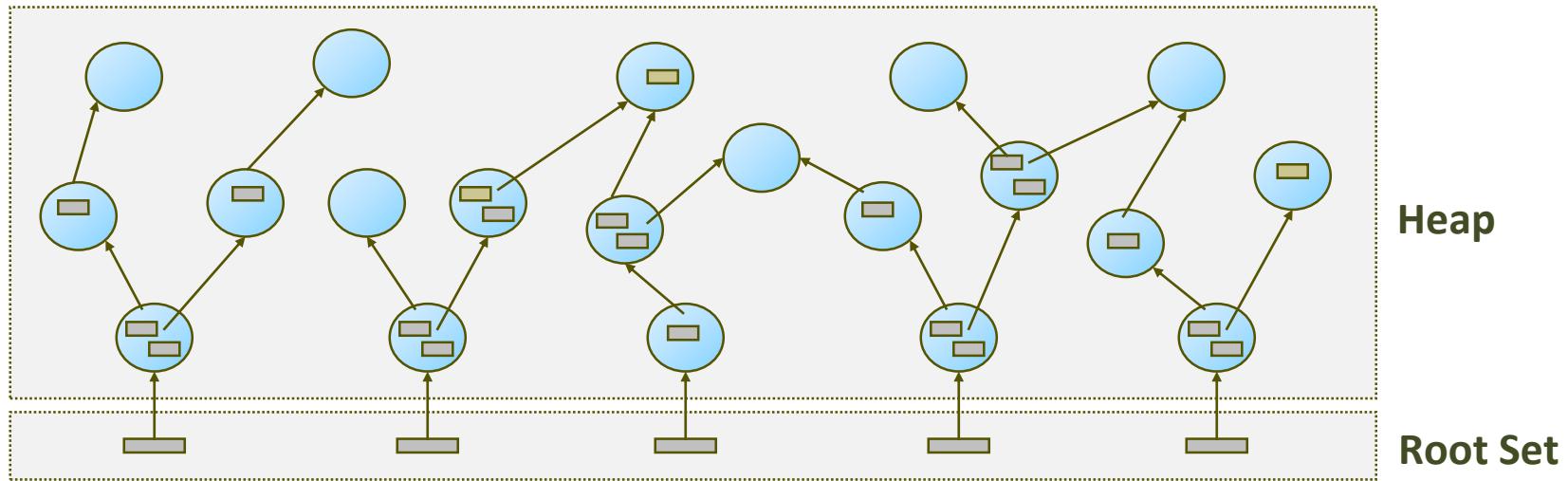
- Allocate memory from OS
- Manage Java allocations
- Clean memory from garbage
 - Garbage
 - Objects that cannot be reclaimed anymore - unreachable
 - Object that are no longer referenced (explicitly & implicitly)
 - Object that are freed via programmatic directives

Memory management

- Reachable objects
 - VM manages a Root Set
 - Consist of :
 - Static references
 - Local references
 - Is the first level of referenced objects
 - Other objects might be referenced by root set objects
 - Garbage collection
 - starts in iterating root set objects
 - Then visiting indirectly referenced objects that in its scope

Virtual Machine

Memory management



- Improves caching of objects & methods
- Fast and fully accurate garbage collection
- Offers more GC algorithms
- Code refactoring
 - Method in-lining
 - Loop unrolling (increasing loop body to reduce iterations)
 - Flow rearranging (fitting compiler instructions order to the machine)

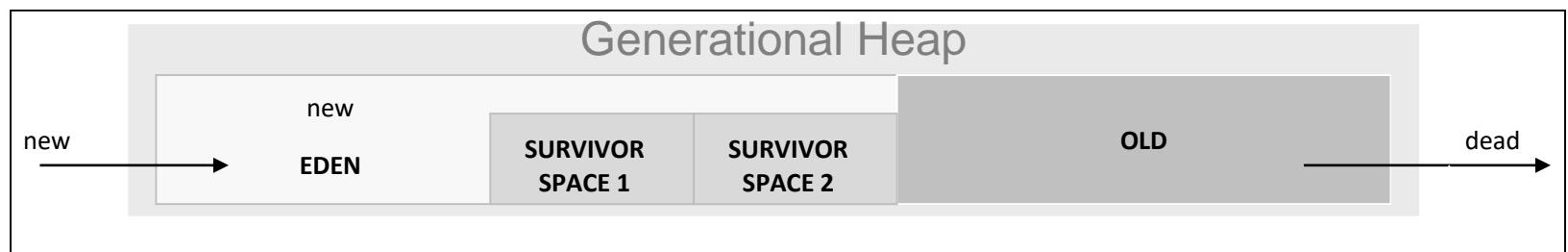
Class Data Sharing

- Share class data between different JVM processes of the same build
- supported from JDK 5
 - Reduce the startup time
 - Reduce memory footprint
- On loading time some system classes metadata is stored in “shared archive” file
- Data is loaded into memory at runtime according to the application needs

<code>java -Xshare:dump</code>	- regenerate file
<code>java -Xshare:off</code>	- disable sharing
<code>java -Xshare:on</code>	- enable sharing
<code>java -Xshare:auto</code>	- default, enable when possible

- Smart thread queue management
 - Hard worker threads will get much more CPU time
- Faster thread synchronization
- Statistics between program executions to improve performance
- Reflection & Serialization improvements
- Advanced logging when handling native crashes specifying:
 - function name
 - library name
 - source-file name
 - line number where the error occurred

- Monolithic heap
 - single heap
 - easy to tune
 - not much to manage & therefore consume less memory
- Generational heap
 - separates the heap area into New objects region & Old object region
 - survivor spaces are sub-regions of New
 - each region is tuned and managed separately



Escape Analysis

- Escaped references
 - When a stack creates an object and hands it to an outside caller
- Non-escaped references
 - When a stack is the only one uses an object
 - When a caller assigns an object to a stack and get rid of his reference

```
public Object method (){
    Object o = new Object();
    otherMethod(o) // Object escaped...
}
```

```
public Object method (){
    Object o = new Object();
    ...
}
```

Escape Analysis

- What is 'escape analysis' ?
 - Tracking non-escape references
 - Turning the non-escape object state to be part of the stack that uses it instead of being allocated on the heap

JVM tracks non-escape references
And turns the object state to be
part of the stack that uses it

```
public Object method (){
    Pixel p = new Pixel(100,30);
    ...
}
```

```
public Object method (){
    int x=100;
    int y=30;
    ...
}
```

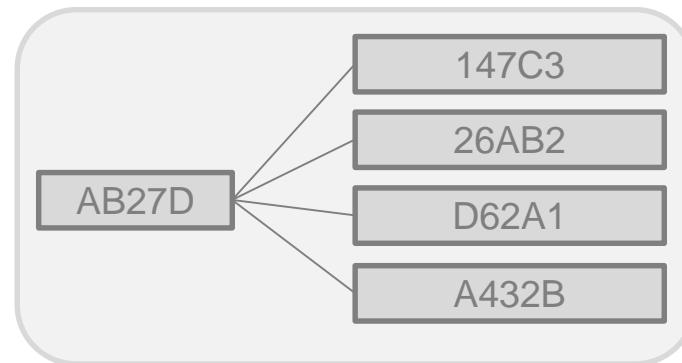


Escape Analysis

- Done during interpretation phase
- Supported in Java SE 6 u23
- To activate use:
 - XX: +DoEscapeAnalysis

Compressed Pointers

- JDK 7.0 improvement
- For reducing JVM process footprint
- Compressed 64-bit ordinary object pointers (oops)
 - Java uses 32 bit offsets + 32 bit for a specific address
 - Offsets are the prefixes of the whole virtual address
 - Instead of : we get:





Smaller Footprint

- Java 8 smaller VM
 - Up to Java 7 footprints averages are
 - 6Mb for –client
 - 9Mb for –server
 - In Java 8 JVM shouldn't use more than 3Mb on startup
 - Smaller plugins are better for small clients....
 - How what it achieved?
 - By getting rid of large components from Java Kernel
 - Kernel makefiles now differentiate between required and optional components



GC

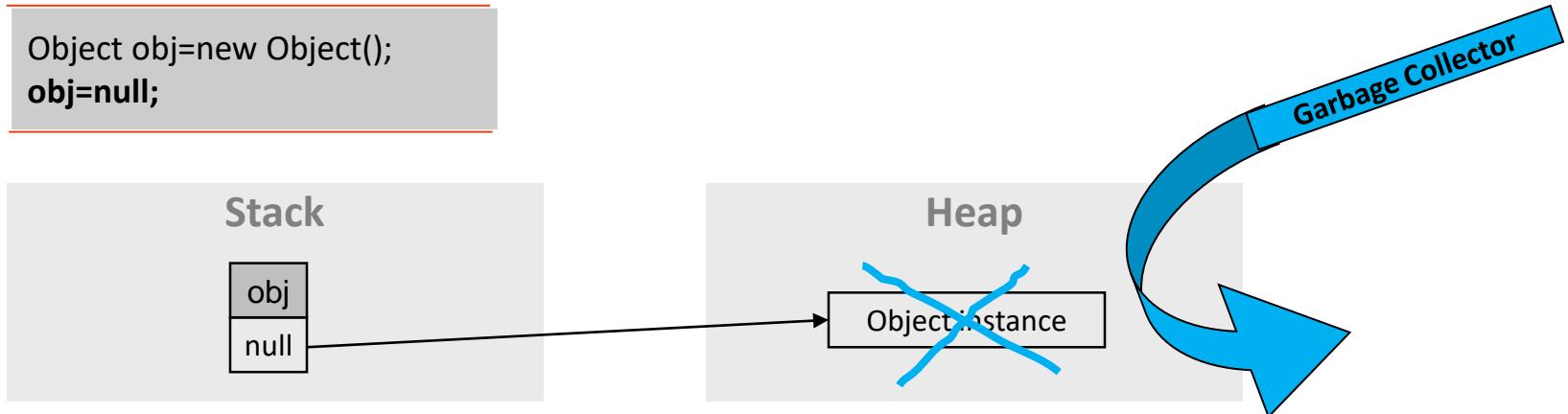


- Introduction
- GC types
- GC Algorithms
- Heap generations
- G1
- Tuning

Garbage Collector

- Responsible for cleaning Objects from the Heap
- Is a low priority daemon thread that runs aside the main program
- Unreachable objects are destroyed [*finalize()* is called]
- Can be invoked by the programmer [*System.gc();*] – not recommended
- Helps in keeping development easy and comfortable

```
Object obj=new Object();
obj=null;
```



Garbage Collector

finalize() method

- Is called by the GC before removing an object from the heap
- Calling it programmatically will not result in the object destruction – not a destructor !
- Do not count on it to perform business logic unless you know what you are doing
 - Since GC is a daemon thread – if main thread dead - GC might not make the call
- Exception thrown are ignored by the CG

gc() method

- Do not count on it to “stop the world” - it doesn’t
- GC gets a higher priority and a better chance to be the next one to get CPU time
- Useful for forcing heap cleanup when there’s a memory leak when debugging

Log GC to a file

- Run application with `-Xloggc:<file>`



Garbage Collector

Roles:

- Mark the object for deletion when applicable
- Remove the object from memory
- All GCs marks objects
- GCs algorithms specifies the way they are removed
- Triggered by:
 - VM allocation failure (that suspends the current thread for the GC)
 - `System.gc()`

Garbage Collector

Monitoring GC

```
java -verbosegc MyProgram
```

```
C:\j2sdk1.4\rony\jobs\fileTree>java -verbosegc -Xmx32m -Xms32m FileManager
[GC 2112K->587K(32576K), 0.0680718 secs]
[GC 2699K->610K(32576K), 0.0165331 secs]
[GC 2722K->641K(32576K), 0.0149122 secs]
```

Amount of freed Kb

Amount of occupied Kb

Heap current size

Time consumed by GC

Garbage Collector

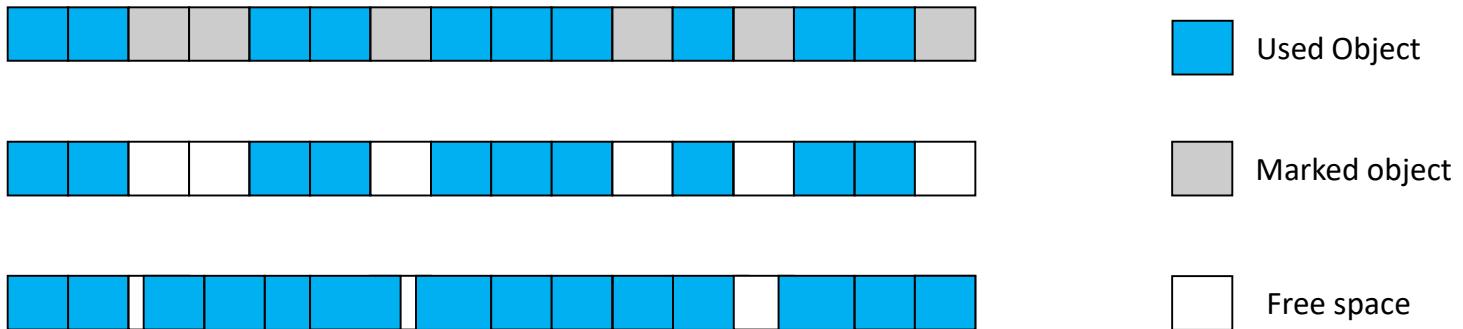
GC Strategies:

- Sweeping GC
- Compacting GC
- Copying GC

Garbage Collector

Sweeping GC

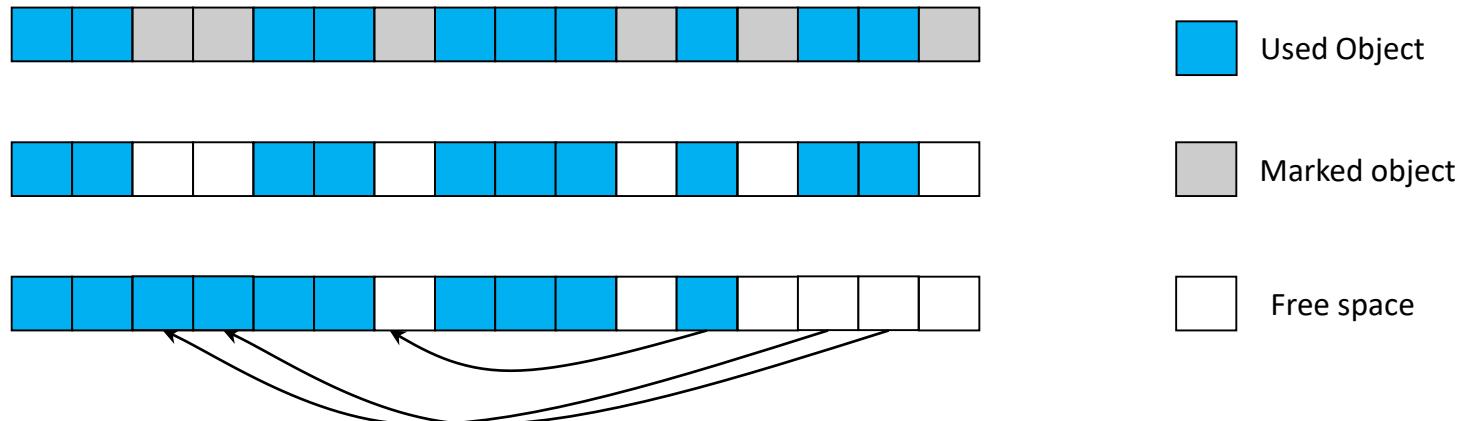
- Scans the heap and simply removes all marked objects
- Quick since no further management takes place
- Leads to free memory fragments making new allocation to become inefficient



Garbage Collector

Compacting GC

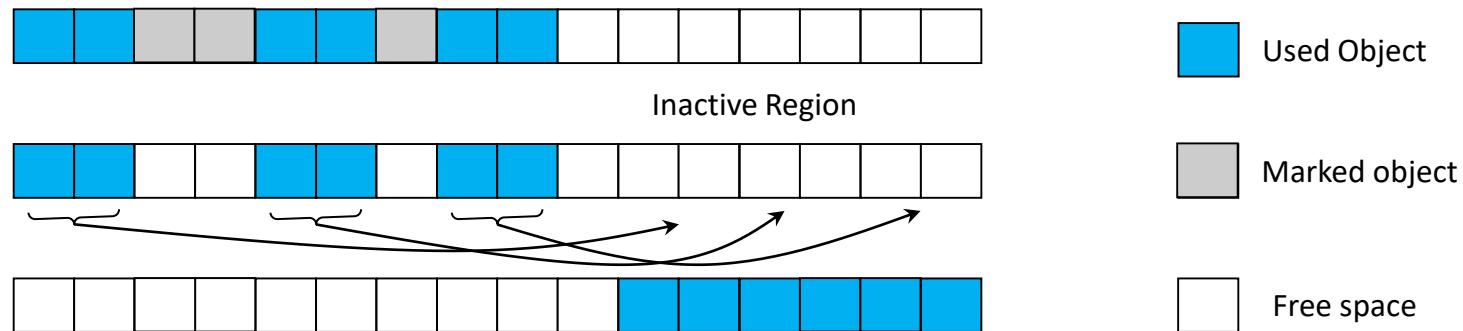
- Scans the heap and removes all marked objects
- Moves used objects to freed memory area to prevent memory fragments
- Needs to check if any moved object matched the free space fragment
- Therefore slower



Garbage Collector

Copying GC

- Copies used objects to inactive regions
- Less need in matching object to available space
- Faster than Compacting CG
- Requires greater heap in order to manage inactive regions



Garbage Collector

GC Types:

- Simple
 - blocks all threads while cleaning – “stop the world GC”
 - More heap allocations for objects causes longer pause time
- Concurrent
 - allows other threads to run along the GC threads
 - each GC thread is responsible for one region in the heap [concurrent marking]
 - available only for –server VM instances
 - use –Xcongc to enable

Garbage Collector

GC Types (cont'):

- Parallel
 - distribute the GC work to multiple processors
 - one thread coordinates a specified number of helper threads
 - available only for –server VM instances
 - one thread per processor - done via –Xgcthreads[n]
- Incremental
 - relevant for Old generation
 - breaks the region into smaller chunks
 - cleans each region individually
 - breaks overall pause time into many short pauses but overall throughput increases
 - use -Xincgc

Generational GC

- Heap generations – New
- Minor GC – Copying GC
- EDEN & 2 survivor spaces
 - intensive allocations
 - intensive GC activity
 - Inefficient memory usage
- When EDEN is about to overflow – copying collection takes place:
 - Most of its objects are unreferenced
 - Those who are referenced are copied to the survivor space 1 inactive region
 - Next time – objects are copied from both EDEN & SP1 to the SP2
 - SP1 is now clean again – so the phases can repeat until –
 - Long lived object eventually moved to OLD region

Generational GC

- Heap generations - Old
 - Less allocations
 - Less GC iterations
 - Efficient memory usage
- OLD region may use several GC strategies
 - Sweep / compact
 - Incremental GC
 - Concurrent GC
 - Parallel GC
- There is no dedicated GC for Old region
 - Full GC (collects on New, Old & Perm)

- G1 (Garbage First)
 - A long run replacement for CMS (Concurrent Mark Sweep GC)
 - Available in Java 6 update 14
- Default GC in Java 9
 - Runs both on New & Old regions

More on G1

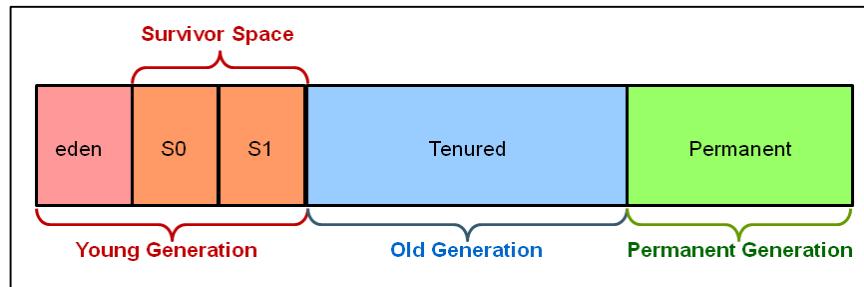
- Instead of sweeping – compacts and defragments
- Also generational (acts on NEW & OLD regions)
- Based on concurrent / parallel behavior
- Like CMS, a low-pause GC – but better
- Usage: `--XX: +UseG1GC`
 - In Java 6 activate feature first with: `-XX:+UnlockExperimentalVMOptions`

Note: G1 increases JVM process footprint due to area collections and management

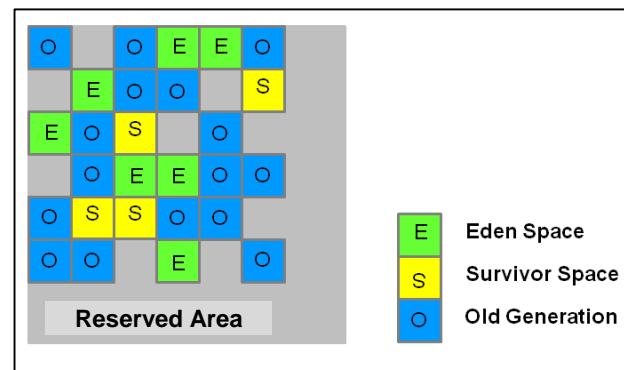
Note: No need to switch to G1 if CMS works fine (low pauses and good throughput)

G1 – How does it work ?

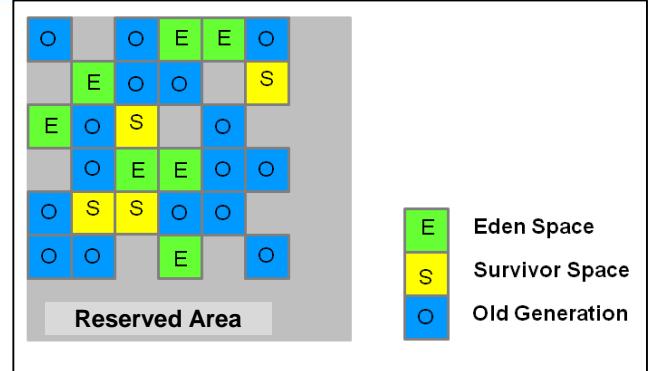
- Previous GCs (serial, parallel, CMS) manages objects in predefined memory regions:



- G1 manages objects like this:



G1 – How does it work ?



- heap is partitioned into a set of equal-sized heap regions
 - Each partition has a role (Eden / Survivor Space / Old)
 - Step1: G1 performs a concurrent global marking of unreachable objects
 - Step 2: G1 tracks areas where most phantom objects exist
 - Step 3: G1 collects in these areas FIRST(!)
 - Step 4: G1 compacts the remaining object in these areas (not much left....)
 - Reserved Area
 - for tenured
 - helpful in cases of huge allocations
 - Size may be set in heap percent

Java 10 – Parallel full G1 CG

- G1 works mostly in an ‘incremental’ way when performing full GC
 - Allows to clean on most reclaimable space blocks first
 - Means that GC never really performs full-GC...
- In some conditions where lots of objects must be cleaned
 - G1 switches to CMS (‘full’ mode) to perform a full GC
 - Before 10: G1 - CMS was single threaded – risk in long pause time
 - From 10: G1 – CMS is parallel (uses the same number of threads set with :GCThreads)

Garbage Collector

Loitering objects

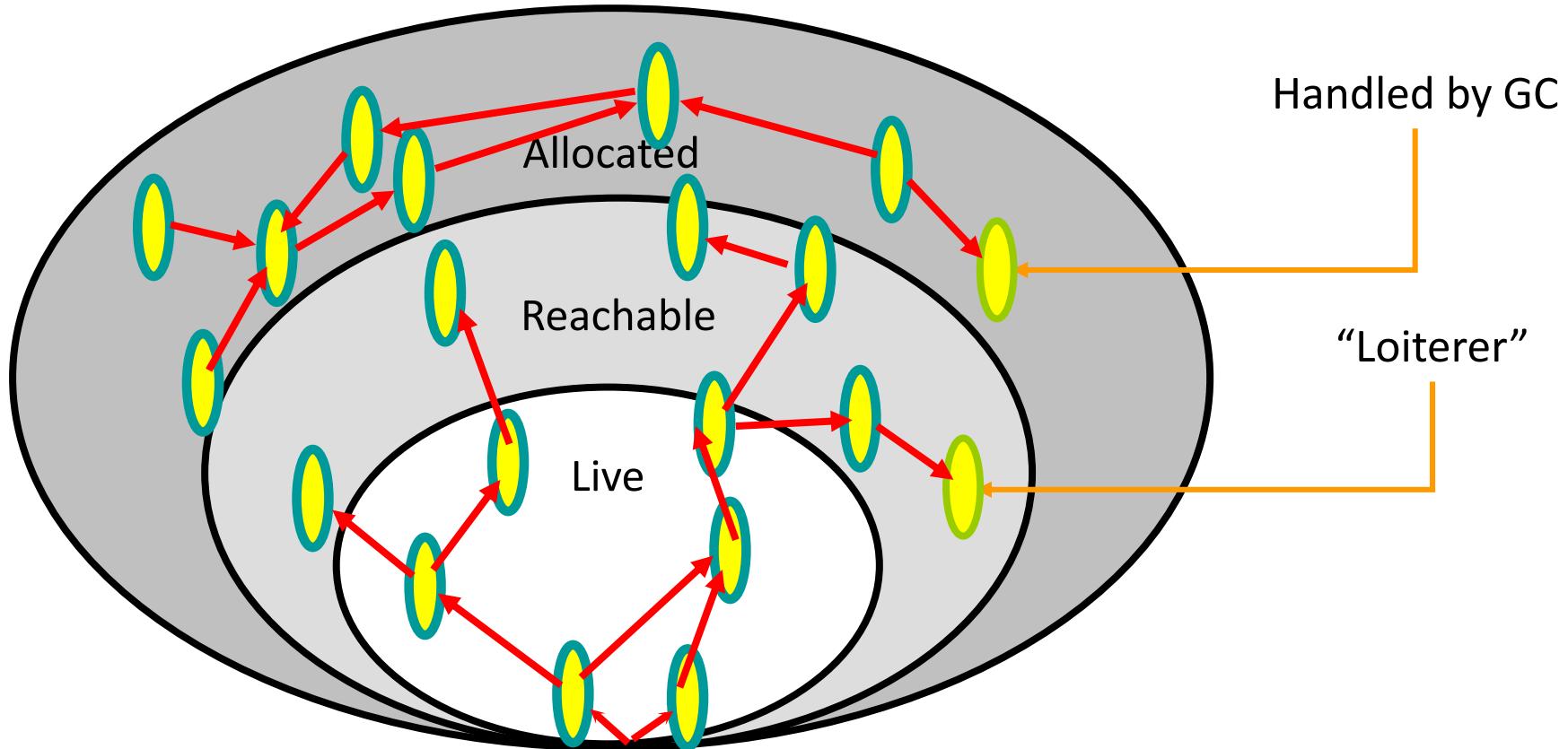
- Are unused referenced objects
- Cannot be garbage collected
- How do we get to this situation ?
 - Mainly by
 - Business flow / use-case / method allocate resources in order to implement some business
 - Operation flow ends successfully but –
 - Resources are not entirely removed

But also by

- Returning a newly created object to a non efficient caller
- Poor variable scope definitions

Garbage Collector

Loitering objects



Garbage Collector

Loitering objects

- Identifying loitering objects
 - Identify problematic use cases
 - Use monitoring & debugging tools to track allocation done in the use case implementation
 - Measure the state before entering the use case flow
 - Look the differences after it ends
 - Are those the new objects you expected
 - Are any unnecessary objects are still referenced
- Fixing loitering objects
 - Track down loitered objects
 - Track the objects that allocated them
 - Determine and implement the points of making them unreachable

Tuning

- Garbage First (G1) Garbage Collection Options – taken from oracle.com

Option and Default Value	Description
-XX:+UseG1GC	Use the Garbage First (G1) Collector
-XX:MaxGCPauseMillis=n	Sets a target for the maximum GC pause time. This is a soft goal, and the JVM will make its best effort to achieve it.
-XX:InitiatingHeapOccupancyPercent=n	Percentage of the (entire) heap occupancy to start a concurrent GC cycle. It is used by GCs that trigger a concurrent GC cycle based on the occupancy of the entire heap, not just one of the generations (e.g., G1). A value of 0 denotes 'do constant GC cycles'. The default value is 45.
-XX:NewRatio=n	Ratio of old/new generation sizes. The default value is 2.
-XX:SurvivorRatio=n	Ratio of eden/survivor space size. The default value is 8.
-XX:MaxTenuringThreshold=n	Maximum value for tenuring threshold. The default value is 15.
-XX:ParallelGCThreads=n	Sets the number of threads used during parallel phases of the garbage collectors. The default value varies with the platform on which the JVM is running.
-XX:ConcGCThreads=n	Number of threads concurrent garbage collectors will use. The default value varies with the platform on which the JVM is running.
-XX:G1ReservePercent=n	Sets the amount of heap that is reserved as a false ceiling to reduce the possibility of promotion failure. The default value is 10. (promotion: from Eden to Survivor space to Tenured...)
-XX:G1HeapRegionSize=n	With G1 the Java heap is subdivided into uniformly sized regions. This sets the size of the individual sub-divisions. The default value of this parameter is determined ergonomically based upon heap size. The minimum value is 1Mb and the maximum value is 32Mb.

Tuning

- Performance Options – taken from oracle.com

Option and Default Value	Description
-XX:CompileThreshold=10000	Number of method invocations/branches before compiling [-client: 1,500]
-XX:MaxHeapFreeRatio=70	Maximum percentage of heap free after GC to avoid shrinking.
-XX:MaxNewSize=size	Maximum size of new generation (in bytes). Since 1.4, MaxNewSize is computed as a function of NewRatio. [1.3.1 Sparc: 32m; 1.3.1 x86: 2.5m.]
-XX:MaxPermSize=64m	Size of the Permanent Generation. [5.0 and newer: 64 bit VMs are scaled 30% larger; 1.4 amd64: 96m; 1.3.1 -client: 32m.]
-XX:MinHeapFreeRatio=40	Minimum percentage of heap free after GC to avoid expansion.
-XX:NewRatio=2	Ratio of old/new generation sizes. [Sparc -client: 8; x86 -server: 8; x86 -client: 12.] -client: 4 (1.3) 8 (1.3.1+), x86: 12]
-XX:NewSize=2m	Default size of new generation (in bytes) [5.0 and newer: 64 bit VMs are scaled 30% larger; x86: 1m; x86, 5.0 and older: 640k]
-XX:ReservedCodeCacheSize=32m	Reserved code cache size (in bytes) - maximum code cache size. [Solaris 64-bit, amd64, and -server x86: 2048m; in 1.5.0_06 and earlier, Solaris 64-bit and amd64: 1024m.]

Tuning

- Refactoring Options – taken from oracle.com

Option and Default Value	Description
-XX:InlineSmallCode=n	Inline a previously compiled method only if its generated native code size is less than this. The default value varies with the platform on which the JVM is running.
-XX:MaxInlineSize=35	Maximum bytecode size of a method to be inlined.
-XX:FreqInlineSize=n	Maximum bytecode size of a frequently executed method to be inlined. The default value varies with the platform on which the JVM is running.
-XX:LoopUnrollLimit=n	Unroll loop bodies with server compiler intermediate representation node count less than this value. The limit used by the server compiler is a function of this value, not the actual value. The default value varies with the platform on which the JVM is running.

Tuning

- Setting Java 8 Metaspace memory attributes

Field	Description
-MaxMetaspaceSize	Sets the maximum size of metaspace memory

- From now on:
 - Native memory is allocated as needed
 - OutOfMemoryError: Metaspace Error
 - Use –verbosegc to view Metaspace GC activity

Java 10 App Class Data SharE

AppCDS

- Class are loaded natively on Metaspace
- When running an application you can
 - Specify classes to be archived – XX:DumpLoadedClassList
 - create an archive – Xshare
- Other applications may use archive and save jars scanning...
 - X:dump:on
- In any case the -XX:+UseAppCDS flag should be specified

Java 10 Heap Allocation

Alternative Heap Allocation

- Java uses DRAM by default
 - DRAM
 - Dynamic random-access memory
- There are strong alternative chips like NVDIMM
 - NVDIMM - non-volatile dual in-line memory module
- In order to specify alternative heap allocation in Java 10:
 - `-XX:AllocateHeapAt=<path>`

Weak References

Basic terms

- Strong reference
 - Regular references
 - GC may remove objects that have no strong references
- Weak reference
 - Special references that are used to reference objects
 - GC may remove objects when all references to that object are weak

Weak References

The need

- When storing data in Collections , for example
 - The collection holds references even when data is no longer required
 - Removing data from the collection might effect clients
- When caching heavy-weight objects
 - Once again – that caching might prevent data from being GC'ed
 - That cache mechanism must be maintained
- Java is capable of managing garbage but:
 - In those cases, when strong references are used, GC is irrelevant
 - We will be flooded with loitered objects & memory leaks

Weak References

The solution

- Give clients strong references
- Manage resource collection & cache via weak references
- Java provides 4 strength levels for referencing
 - Strong reference
 - Soft reference
 - Weak reference
 - Phantom reference

Weak References

java.lang.ref.Reference

- Is the super class for all reference types
- Has 3 subclasses for each weak reference level
 - Each subclass effect the way CG treats objects
 - Each subclass may be extended for more functionality
- Has the following operations:
 - clear – clears the object held in the Reference object
 - get – returns a strong reference to the object held in the Reference
 - enqueue – adds the Reference to a Reference Queue (if registered to any)

Weak References

Reference Queue

- Queue that holds references to null
 - Reference that is no longer points to an object is added to that queue
 - Reference must register itself to a queue in order to gain this service
 - Is an automatic mechanism
- Registering a reference to a queue is done via Reference constructor
- References can be consumed via 3 of the queue methods
 - Poll – returns the last added reference or null if empty
 - Remove – returns the last added reference or blocks until one becomes available
 - Remove (timeout) – same as previous but might return null when timed out

Weak References

Soft reference

- Objects with soft references will be lazily garbage collected
 - Means that the system will let them survive GC even with no strong references
 - System cleans them when any memory limitation occurs
- Great for cache auto-management
 - Clients uses strong references to heavy weight objects
 - Cache services uses soft references
 - When clients release their strong references – cached data is not GC'ed
 - JVM requirement for more memory may lead to cached object removal
 - Timing is determined according to client use & system resources automatically !
- Use *java.lang.ref.SoftReference*

Weak References

Soft reference - example

Client gets strong reference - word

Cache mechanism gets soft reference

Client release reference when no longer needed

Cache keeps on referencing even between GC calls

Other clients may ask for strong references via soft.get() method

```
import java.lang.ref.SoftReference;  
  
public class SoftExample {  
  
    public static void main(String[] args) {  
        Object obj=new Object();  
        SoftReference<Object> soft = new SoftReference<Object>(obj);  
        obj=null;  
        System.gc();  
        try{Thread.yield();}catch(Exception e){}  
        System.out.println(soft.get());  
    }  
}
```

Output:
java.lang.Object@7d772e

Weak References

Weak reference

- Objects with weak references will be eagerly garbage collected
 - Means that they will be removed on the first GC iteration
 - Will stay alive if strong and soft references still exist
- Great for cache resource collections
 - Clients uses strong references to resource objects
 - Containers may collect those objects via weak references for:
 - Monitoring & logging
 - Managing
 - Attach more information and data to each resource while alive
 - When clients release their strong references – resources are GC'ed
 - Container enjoys immediate removal of resources automatically !
- Use `java.lang.ref.WeakReference`

—

Weak References

Weak reference - example

Client gets strong reference - word

Cache mechanism gets weak reference

Client release reference when no longer needed

Containers get rid from weak refs when no strong reference exist

Other clients may ask for strong references via soft.get() method

- If objects still alive – a strong reference is returned
- If objects were garbaged – a null value is returned

```
import java.lang.ref.SoftReference;  
  
public class SoftExample {  
  
    public static void main(String[] args) {  
        Object obj=new Object();  
        WeakReference<Object> weak = new WeakReference<Object>(obj);  
        obj=null;  
        System.gc();  
        try{Thread.yield();}catch(Exception e){}  
        System.out.println(weak.get());  
    }  
}
```

Output:
null

Weak References

Phantom reference

- Objects becomes phantom referenced when
 - No strong or soft references exist &
 - The object was finalized &
 - Only phantom references exists, or none
- Main usage is to add functionality to the origin object finalize() method
 - Offers more flexible mechanism – event queue
- Phantom referenced objects are already finalized & cannot reproduce strong references (unlike soft & weak references)
 - Therefore get() method always returns null
- Use `java.lang.ref.PhantomReference`

Weak References

Phantom reference - example

```
import java.lang.ref.*;
import java.util.Date;

public class PhantomExample {

    public static void main(String[] args) {
        Object obj=new Object();
        ReferenceQueue<Object> queue = new ReferenceQueue<Object>();
        PhantomReference<Object> phantom = new PhantomReference<Object>(obj,queue);
        obj=null;
        System.gc();
        try{Thread.yield();}catch(Exception e){}
        try {
            queue.remove();
            // Now the object was finalized for sure
        } catch (InterruptedException e) {}
    }
}
```

Creating phantom reference & register in queue

Client release ref when no longer needed

Reference is added to the queue after obj is finalized but before its memory reclaimed

Remove operation blocks until a ref is available in the queue

Output:
Tue Jan 22 13:54:11 GMT 2008



Class Loading



- Introduction
- Endorsed, Ext and Classpath
- Build-in mechanism
- Custom class loading
- Multithreaded CL

Class Loading

- Instances of class `java.lang.Class` represent classes and interfaces in a running Java application
- Class has no public constructors
 - Any applications cannot instantiate it
 - Class is automatically instantiated by JVM/class loader, when a class loading process is done
- Each instance of Class keeps a reference to a class loader that created it

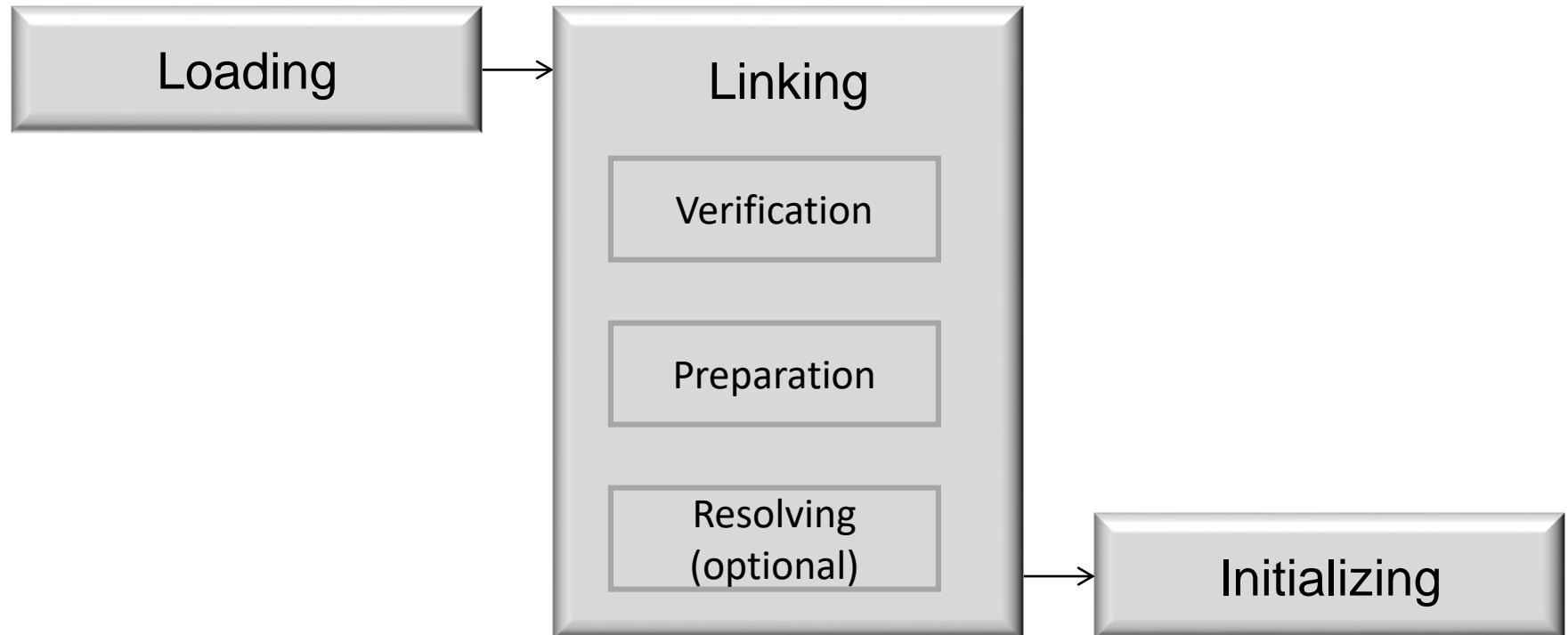
Class Loading

- When you instantiate a new object `new Foo()` class loader performs the following major steps:
 - Loading – loading class byte data from a file or other
 - Linking – preparing the binary data to be used in Java runtime
 - Initializing – loading and populating class variables and blocks
- If those 3 steps performed successfully – Foo instance can be created & returned
- Else – exceptions are thrown (later)

Class loading

Class Loading

Class loading process:



Class Loading

- Loading and linking phases are flexible on their time of execution
- Initializing, on the other hand, may occur only in these active uses:
 - When creating new instance for the first time
 - Invocation of a static method on a class
 - The use of a static field in a class
 - The usage of reflection upon a class (using Method required its Class..)
 - The initialization of a subclass of a class
 - On launch – loading the class with main() method

Class Loading

Classes vs. Interfaces

- When classes are loaded -
 - All its super-classes must be loaded too
 - Interfaces are not loaded
- When interfaces are loaded -
 - None of its super-interfaces is loaded
- What triggers Interface loading for a class ? - A polymorphic usage of it !

```
public class Airplane extends Vehicle
    implements Flyable {  
    ...  
}
```

```
...  
Airplane plane = new Airplane(); ←  
plane . drive ();  
plane . accelerate ();  
((Flyable) plane ) . fly(); ←  
...
```

Here both Vehicle and Airplane classes are loaded

This line triggers Flyable class loading

Class Loading

Loading phase

- in order to load a class JVM must:
 - take a fully qualified class name
 - parse the class byte stream into the method area*
 - stream can be taken from anywhere (FS, ZIP, DB, Network, Runtime...)
 - create an instance of `java.lang.Class` for the type

*VM Method Area

- A storage area for class information and structure
- Class data is constants, variables, method and constructor code
- Created on VM startup
- This area is sharable among different VM instances (since JDK5 &if share enabled)
- When full – *OutOfMemoryError* is thrown

Linking phase – Verification

- Scan for defects like invalid operation code and invalid signatures
- This phase ensures that:
 - the type obeys the semantics of the Java language
 - VerifyError is thrown at this phase if class is given in a wrong format
 - it won't violate the integrity of the virtual machine
 - LinkageError is thrown when dependant classes were incompatibly changed
 - Has specific sub-errors like AbstractMethodError if there is a call to an abstract method in a class

Linking phase – Preparation

- Allocation for `java.lang.Class` instance
- During this phase the class variables are allocated
 - They are populated with default values just like any other object
 - Initiation values are not applied yet
- More custom adjustments can also be used
 - like method table – table that holds pointers to the data for each method (including data from superclasses)

Class Loading

Linking phase – Resolution

- Locate fields, methods, classes and interfaces symbolically referenced from the class runtime constant pool*
- Optional – if all or some are already located by previous class loadings – no need to do it again..

*Runtime constant pool

- A per type symbol table
- Uses tags bounded with values [class=7, Fieldref=9, Methodref=10, String=8, Integer = 3, ...]
- Created symbolically on loading phase
- Name of each tag can be seen with the Class.getName() method:

```
String → java.lang.String
MyClass → com.my.custom.MyClass
int [] → [I
int [][] → [[I
String [] → [Ljava.lang.String
```

Class Loading

Initialization phase

- Applying all programmers static setup
 - overriding default values with programmer values
 - executing static blocks keeping the order it appears in the class
- Actually, all static initializations and static blocks are gathered into a special method.
 - called ‘initialization method’ or ‘the () method’

```
public class StaticInitializationExample {  
  
    private static double x = Math.random();  
    ...
```

```
public class StaticBlockExample {  
  
    private static double x;  
    static{  
        x = Math.random();  
    }  
    ...
```

Class Loading

VM Default class loaders

- Are created automatically on VM startup
- Bootstrap class loader
 - Loads the standard libraries
 - Classes located at `%JAVA_HOME%/jre/lib/`
 - Also loads `%JAVA_HOME%/jre/lib/rt.jar`
 - From Java 6 – first loads jars placed in the lib/endorsed directory (allows to override Sun's impl.)
- Extension class loader
 - Default implementation Loads the set of extension libraries
 - Classes located at `%JAVA_HOME%/jre/lib/ext/`
 - Can be implemented in vendor specific way
- System class loader
 - Loads user defined classes
 - Classes configured and mapped at VM `%CLASSPATH%` (both directories of classes & jars)
 - If class is not found – throws *ClassDefNotFoundException*

Class Loading

Custom class loaders

- Are user implementation of class loading
- Are created programmatically after VM startup
- Implicit or explicit child of the System class loader
- Are forbidden in Applets (too powerful)

- Why use custom class loaders?
 - For customized class loading policy (like DB or Network class loaders)
 - Security and versioning filtering
 - Byte code formatting / adjusting
 - Deploying new modules while previous are still loaded

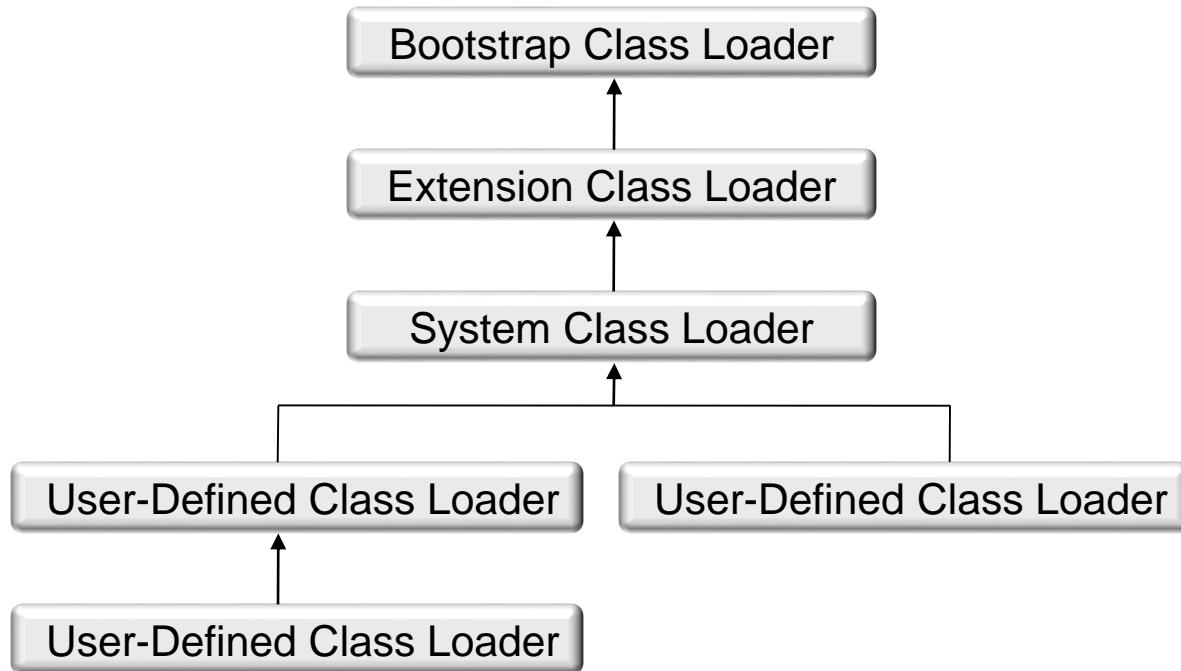
Class Loading

Endorsed directory

- Place mature jars to override immature/buggy implementations in JDK core
- List of API s that currently can be overridden:
 - Java API for XML Processing (JAXP) 1.4.5
 - Java Architecture for XML Binding (JAXB) 2.2.4-1
 - Java Compiler API 1.0
 - Pluggable Annotation Processing API 1.7
 - Common Annotations for the Java Platform 1.1
 - Scripting for the Java Platform 1.0
 - SOAP with Attachments API for Java (SAAJ) (package *javax.xml.soap*) 1.3.9

Class Loading

Class loader hierarchy



Class Loading

Working with custom class loaders

- Coding rules
 - Must extend system class loader (`java.lang.ClassLoader`) or other custom class loader
 - Must override Class `findClass (String name)` method
 - Defines the main logic of how and where the binary data is taken
 - Use `super.defineClass(..)` in order to reflect the loaded byte array into a Class [may also be overridden]
- Using it
 - Create an instance of the custom class loader
 - Use `loadClass (String className, boolean resolve)` method to load a class reflection [Class]
 - Call `Class.newInstance()` method in order to create an object from the loaded class
 - Relevant for classes with empty constructor – otherwise use reflection to call the constructor with appropriate parameters
 - Causes static block execution, default values assignment & constructor execution

Class Loading

Working with custom class loaders

- Example:

```
public class CustomClassLoader extends ClassLoader {  
  
    public Class findClass (String className){  
        byte[] data=null;  
        //fetch class and load into the byte array  
        //(or load class directly in Class format)  
        return defineClass(className,data,0,data.length);  
    }  
}
```

```
public class Test {  
  
    public static void main(String[] args) {  
        CustomClassLoader loader=new CustomClassLoader();  
        try {  
            Object obj=loader.loadClass("Foo").newInstance();  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
}
```



Class Loading

Replacing default System class loader

- Create a custom class loader
- Don't place class in your classpath... (use ext)
- Set System property with fully qualified name of your class loader
 - Property name: "java.system.class.loader"
 - Property value:"fully.qualified.custom.MyClassLoader"
- If this property is not set – default System CL is used

Class Loading Deadlock

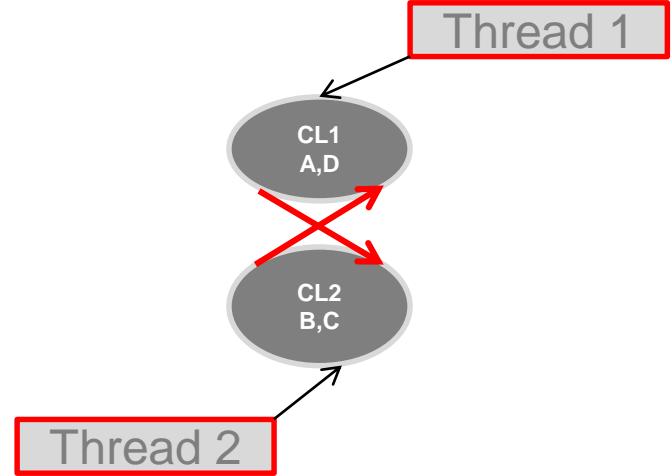
Multithreaded class loading CL

- How CL usually works?
 - Each CL has a parent (except Bootstrap CL)
 - When a class is required, the CL first check on its parent
 - If not loaded yet, CL asks its parent to load it
 - If cant be loaded from parent – loaded by CL
- Note ! Loading classes operation is synchronized

Class Loading Deadlock

Multithreaded class loading CL

- The problem:
 - In this scenario
 - D extends C and B extends A
 - CL1 loads A and D directly.
 - CL2 loads B & C directly.
 - A deadlock situation:
 - Thread1 wants B from CL2
 - Thread2 wants D from CL1



Class Loading Deadlock

- 2 different threads
- 2 different class classes with no direct relations
- 2 separate CLs
- Why do we get deadlock ?!

- Thread1 wants B from CL2
 - It locks CL2 to load B directly – but then –
 - CL2 locks CL1 in order to define B (so CL1 loads A)

- Thread2 wants D from CL1
 - It locks CL1 to load D directly – but then –
 - CL1 locks CL2 in order to define D (so CL2 loads C)

Class Loading Deadlock - Fixed

Multithreaded class loading CL – fixed in JDK 7

- Synchronization is no longer done according to CL
- It is applied to the combination of CL + class name
- Back to our scenario:
 - Thread1 will lock CL1 to load parent class A (lock CL1+"A")
 - Then will lock CL2 to load B (lock CL2+"B")
 - Thread2 will lock CL2 to load parent class C (lock CL2+"C")
 - Then will lock CL1 to load D (lock CL1+"D")
- Problem solved – we have 4 different locks instead of 2...

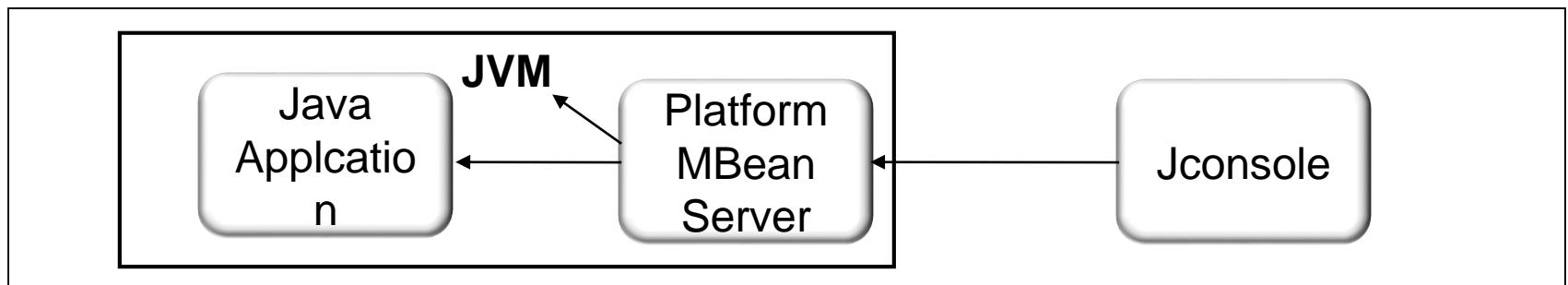
Class Loading Deadlock - Fixed

Multithreaded class loading CL

- How to use it ?
 - Basically this feature is turned off for backward compatibility
 - To turn on – call this method on your custom CL:
 - *registerAsParallelCapable()*
 - If the custom CL overrides *findClass(...)*, then no further changes are needed
 - If the custom CL overrides *loadClass(...)*
 - Method *defineClass()* must be invoked ONCE for any CL & class name pairs
 - Define class obtains a lock on the parent CL to fully load class structure

A new tool provided as part of the JDK

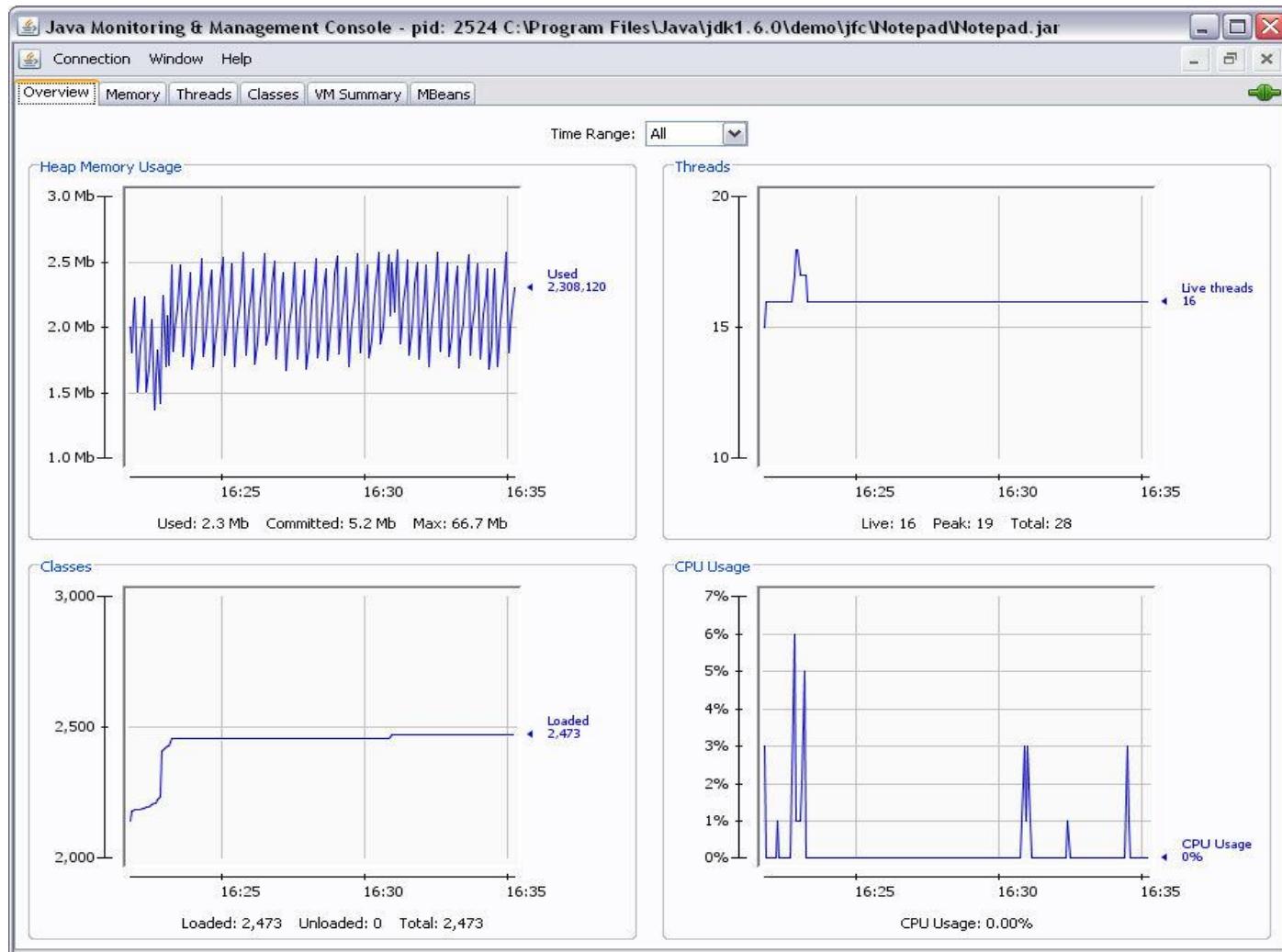
- Provides out-of-the-box monitoring and management support
- Support both remote and local working modes
- Is JMX based
- Provided functionalities:
 - Detect low memory
 - Enable / Disable GC & class loading verbose tracing
 - Detect deadlocks
 - Control log level [where logger is in use]
 - View application's MBeans



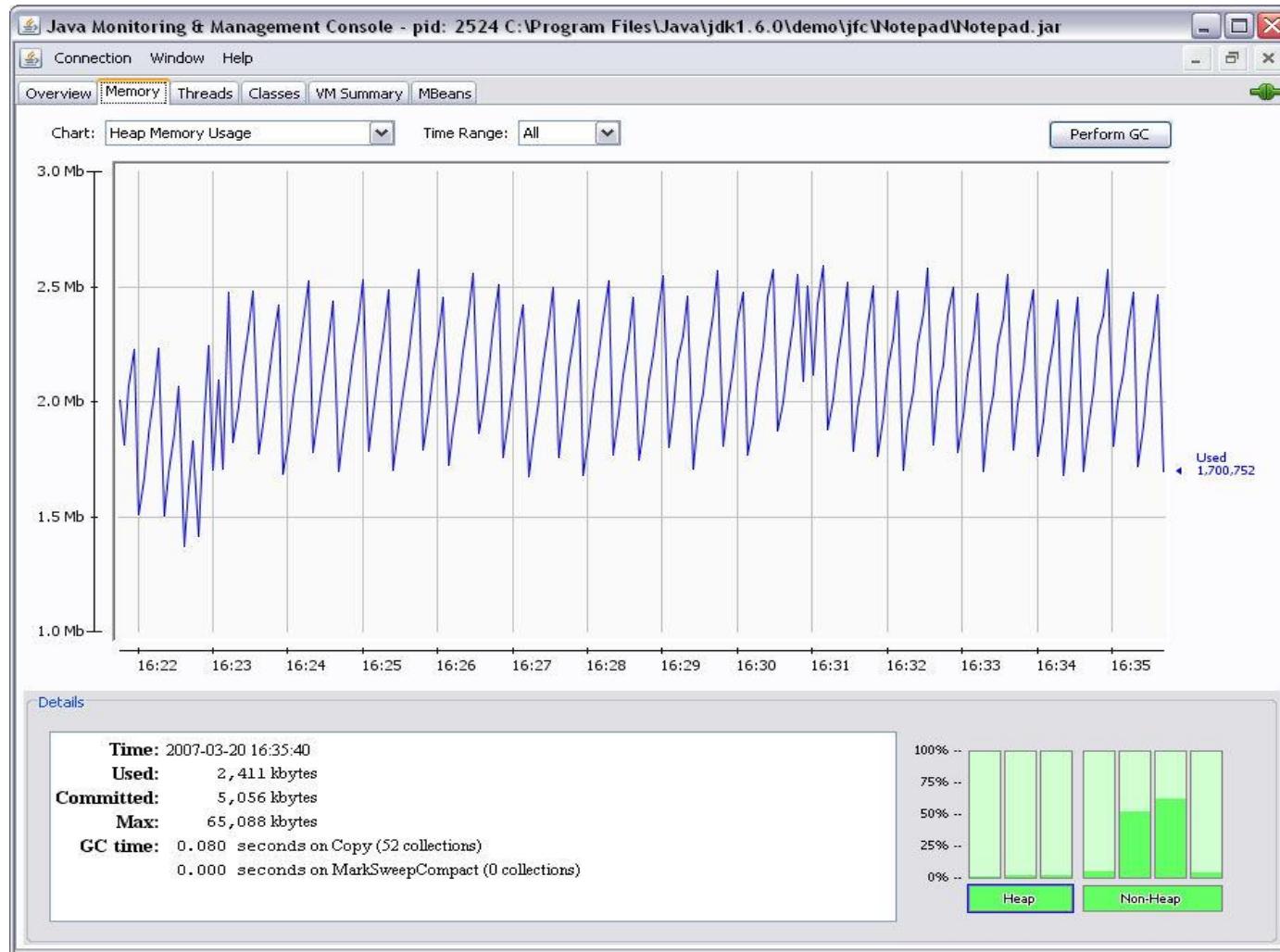
Starting the console

- Locally:
 - Start your application with the system property –
Dcom.sun.management.jmxremote
 - This will activate the Mbean server
 - simply run ‘jconsole’ from the JDK bin directory
 - Choose your application to monitor from list
- Remotely:
 - Start your application with the same property as in local mode
 - Set port via –*Dcom.sun.management.jmxremote.port=portNum*
 - Password can be defined, loaded from file and disabled via additional properties
 - Run jconsole host:port and choose your application from list

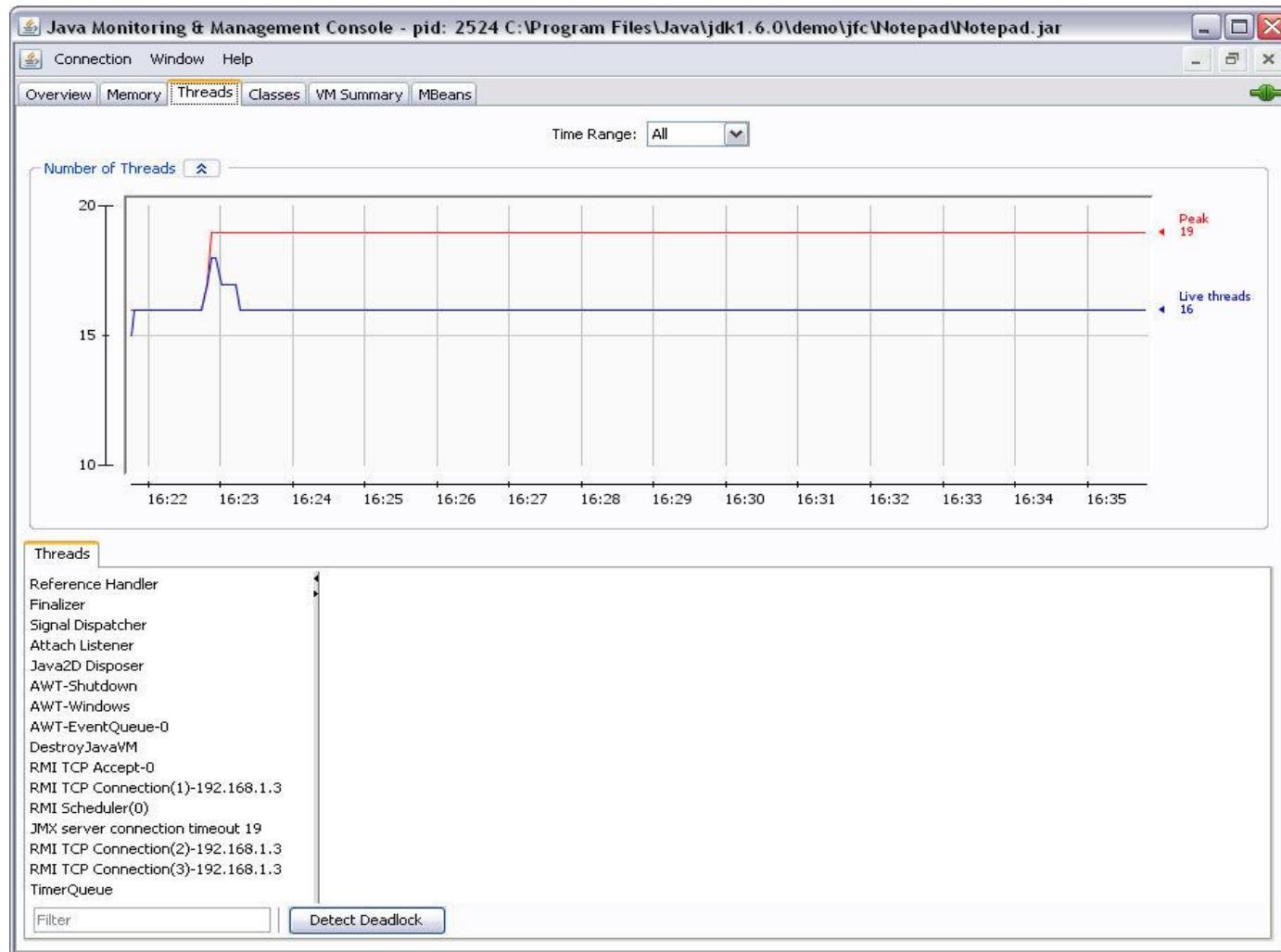
Jconsole



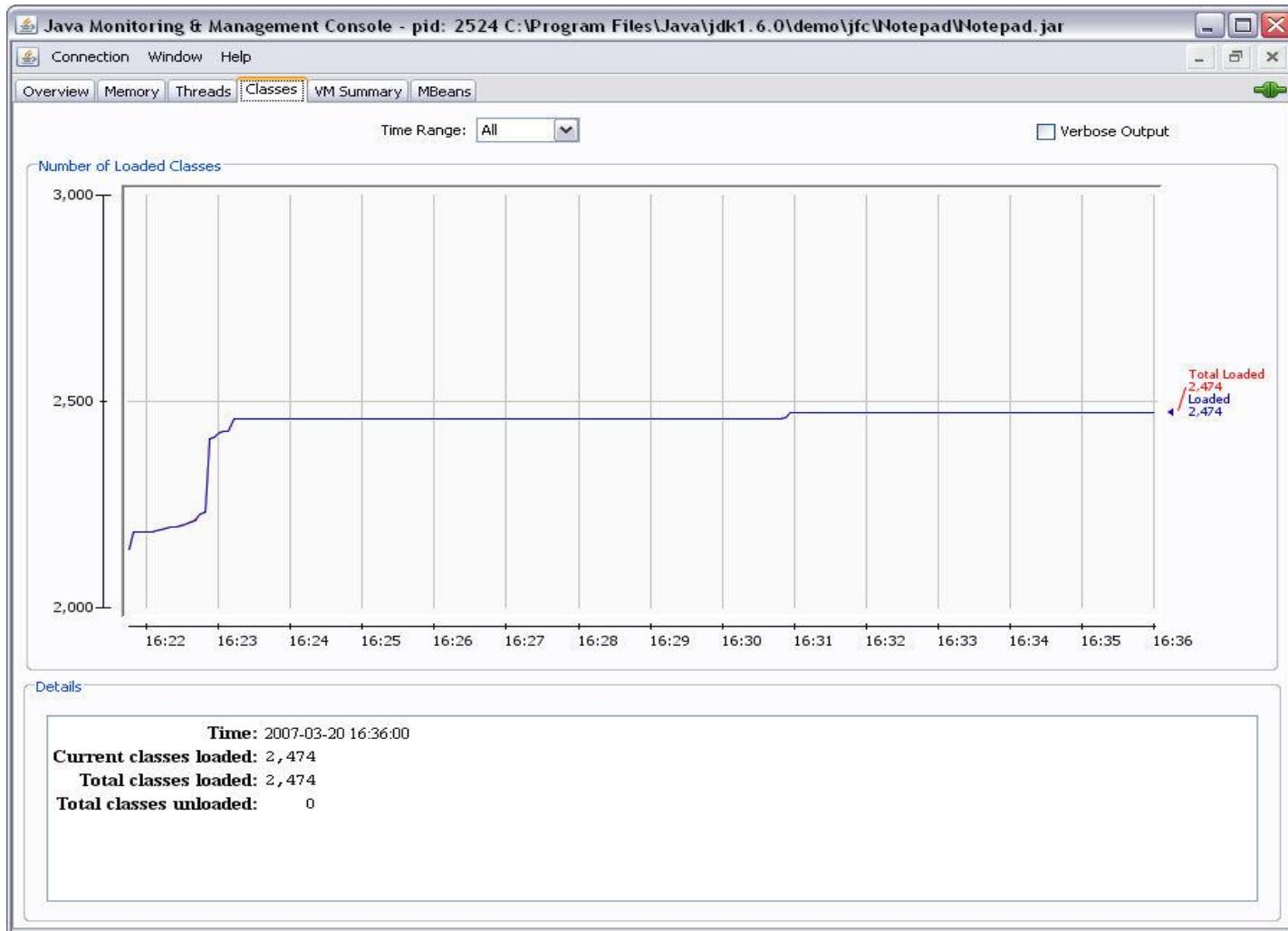
Jconsole



Jconsole



Jconsole



Jconsole

Java Monitoring & Management Console - pid: 2524 C:\Program Files\Java\jdk1.6.0\demo\jfc\Notepad\Notepad.jar

Connection Window Help

Overview Memory Threads Classes VM Summary MBeans

VM Summary
Tuesday, March 20, 2007 4:36:08 PM IST

Connection name: pid: 2524 C:\Program Files\Java\jdk1.6.0\demo\jfc\Notepad\Notepad.jar	Uptime: 14 minutes
Virtual Machine: Java HotSpot(TM) Client VM version 1.6.0-b105	Process CPU time: 6.062 seconds
Vendor: Sun Microsystems Inc.	JIT compiler: HotSpot Client Compiler
Name: 2524@home	Total compile time: 0.618 seconds

Live threads: 16	Current classes loaded: 2,474
Peak: 19	Total classes loaded: 2,474
Daemon threads: 13	Total classes unloaded: 0
Total threads started: 28	

Current heap size: 1,945 kbytes	Committed memory: 5,056 kbytes
Maximum heap size: 65,088 kbytes	Pending finalization: 0 objects

Garbage collector: Name = 'Copy', Collections = 53, Total time spent = 0.080 seconds	
Garbage collector: Name = 'MarkSweepCompact', Collections = 0, Total time spent = 0.000 seconds	

Operating System: Windows XP SP1	Total physical memory: 1,048,044 kbytes
Architecture: x86	Free physical memory: 576,368 kbytes
Number of processors: 2	Total swap space: 2,507,516 kbytes
Committed virtual memory: 32,776 kbytes	Free swap space: 2,004,340 kbytes

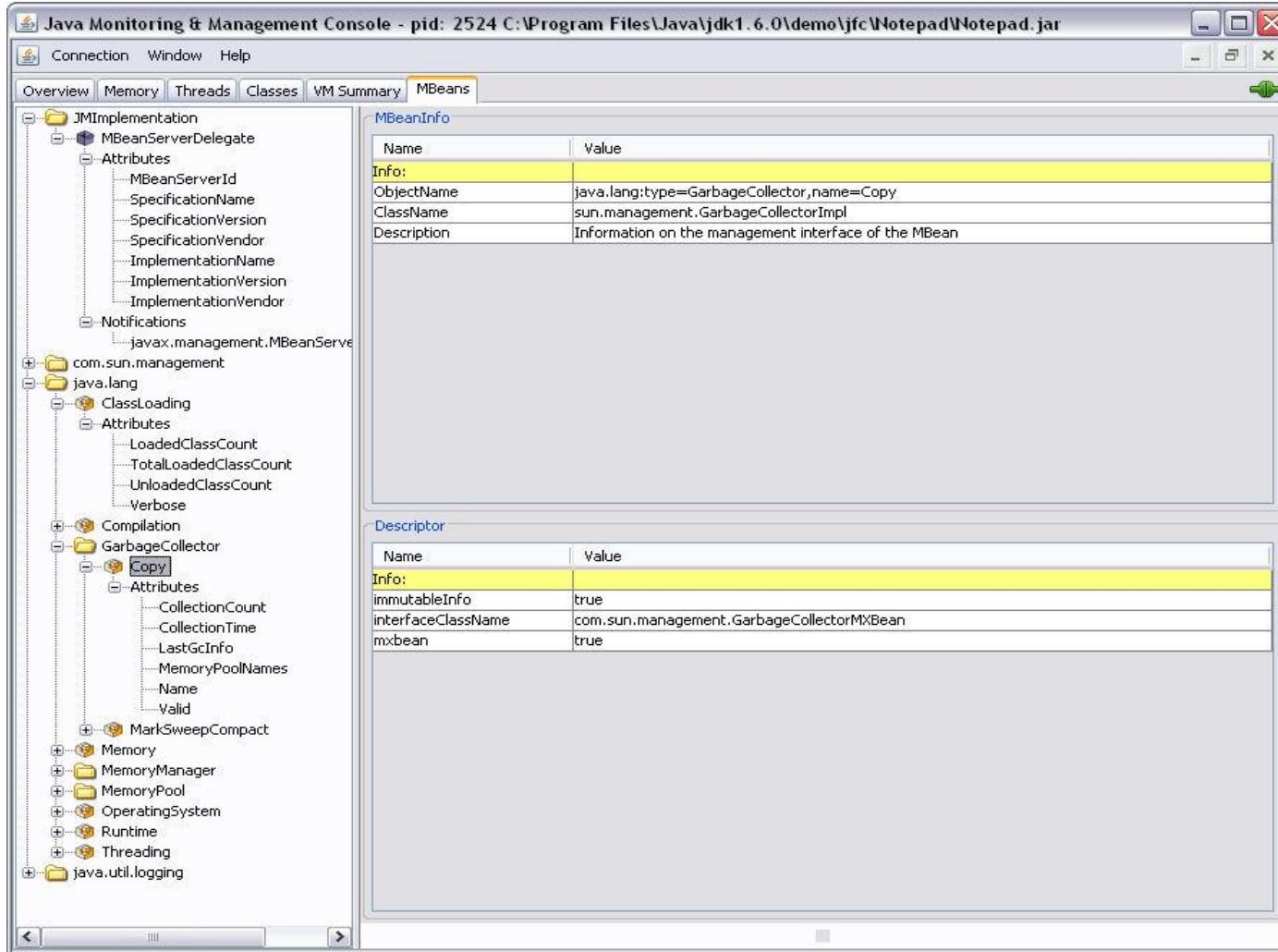
VM arguments:

Class path: C:\Program Files\Java\jdk1.6.0\demo\jfc\Notepad\Notepad.jar

Library path: C:\Program Files\Java\jre1.6.0\bin;;C:\WINDOWS\SunJava\bin;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\WBem;C:\Program Files\ATI Technologies\ATI.ACE;C:\Program Files\Java\jdk1.6.0\bin

Boot class path: C:\Program Files\Java\jre1.6.0\lib\resources.jar;C:\Program Files\Java\jre1.6.0\lib\rt.jar;C:\Program Files\Java\jre1.6.0\lib\unrsasign.jar;C:\Program Files\Java\jre1.6.0\lib\jsse.jar;C:\Program Files\Java\jre1.6.0\lib\jce.jar;C:\Program Files\Java\jre1.6.0\lib\charsets.jar;C:\Program Files\Java\jre1.6.0\classes

Jconsole



Name	Value
Info:	
ObjectName	java.lang:type=GarbageCollector,name=Copy
ClassName	sun.management.GarbageCollectorImpl
Description	Information on the management interface of the MBean

Name	Value
Info:	
immutableInfo	true
interfaceClassName	com.sun.management.GarbageCollectorMXBean
mxbean	true

Java Mission Control

- Available in the Oracle JDK since Java 7u40
- Originates from JRockit JVM
- Used for 2 main purposes:
 - Monitoring the state of multiple running Oracle JVMs
 - Java Flight Recorder dump file analysis
- Built on JVM JMX Server
- JAVA_HOME\bin\jmc.exe



Java Mission Control

- Real time processing
 - Heap
 - CPU
 - GC Pause time



Java Mission Control

- Event triggers
 - Act when JMX counter exceeds any pre-set limit in any given time

The screenshot shows the Java Mission Control interface with the 'Triggers' tab selected. On the left, a tree view lists trigger rules categorized by JVM. A specific rule under 'Java SE' is selected, showing its details in the main pane.

Trigger Rules

Add trigger rules and activate/deactivate them. Triggers that are not available in the monitored JVM are [disabled](#).

- Java SE
 - CPU Usage - JVM Process (Too High)
 - CPU Usage - JVM Process (Too Low)
 - CPU Usage - Machine (Too High)
 - CPU Usage - Machine (Too Low)
 - Deadlocked Threads
 - Live Set (Too Large)
 - Monitored Deadlocked Threads
 - Process CPU usage - another copy
 - Thread Count (Too High)
- WebLogic Server 10.3 - Examples Server
 - Memory Pressure (Too High)
 - Open Sessions (Too Many)
 - Pending JMS Messages (Too High)
 - Pending Queued Requests (Too Many)
 - Primary Objects (Too Many)
 - Requests Waiting for DB Connection (Too High)
 - Server Health (Not OK)
 - Server State (Not running)
 - Threads Waiting for Bean (Too Many)

Rule Details

Condition | Action | Constraints

Description: Process CPU usage - another copy

MBean Path: java.lang:type=OperatingSystem

Attribute Name: ProcessCpuLoad

Current Value: 1.56 %

Max trigger value: 0 < 100 %

Sustained period: 1 s

Limit period: 5 s

Trigger when condition is met.

Trigger when recovering from condition

Action

- Application alert
- Console output
- Dump Flight Recording
- HPROF Dump
- Invoke Diagnostic Command
- Log to file
- Send e-mail
- Start Continuous Flight Recording
- Start Time Limited Flight Recording

Java Mission Control

- Memory tab
 - Heap generation
 - Minor & full GC Info

Class	Instances	Size	Delta
char[]	138477	12.4 MiB	753 KiB
byte[]	3211	5.47 MiB	-24.1 KiB
String	134678	3.08 MiB	205 KiB
java.util.HashMap\$Node	94878	2.9 MiB	226 KiB
Object[]	39432	2 MiB	25.6 KiB
java.util.HashMap\$Node[]	28809	1.86 MiB	63.9 KiB
Class	17065	1.86 MiB	216 B
java.util.HashMap	36045	1.65 MiB	32.8 KiB
int[]	17841	1.08 MiB	24.9 KiB

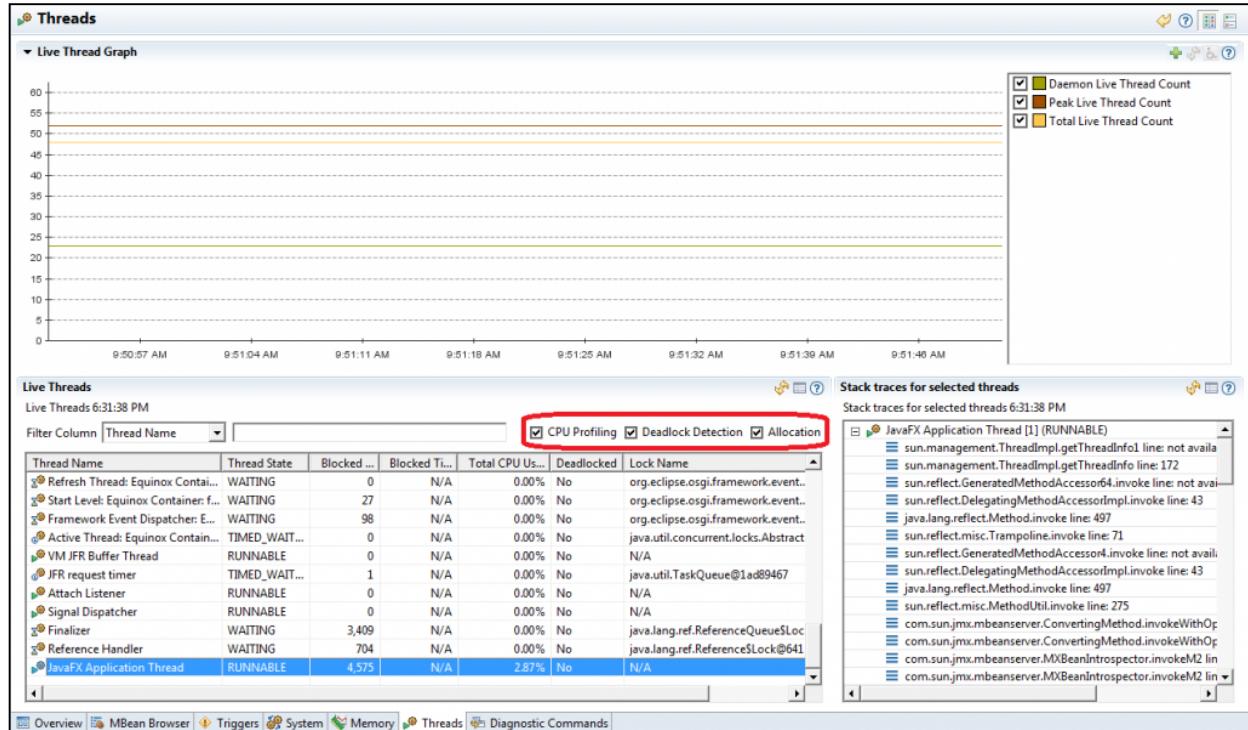
Name	Value	Type	Update Interval	Description
Total Collection Time	20 s 465 ms	Duration	Default	The accumulated collection time.
Collection Count	590	Number	Default	The total number of collections that have o...
GC Start Time	4 d 19 h	Duration	Default	The start time of this GC since the JVM was ...
GC End Time	4 d 19 h	Duration	Default	The end time of this GC since the JVM was ...
GC Duration	19 ms	Duration	Default	The elapsed time of this GC.
GC ID	590	Number	Default	The identifier of this GC which is the numb...
GC Thread Count	10	Number	Default	The number of GC threads.

Pool Name	Type	Used	Max	Usage	Peak Used	Peak Max
Code Cache	NON_HEAP	77.91 MB	240.00 MB	32.46%	78.33 MB	240.00 MB
G1 Old Gen	HEAP	93.59 MB	970.00 MB	9.65%	522.54 MB	970.00 MB
G1 Survivor Space	HEAP	3.00 MB	N/A	N/A	16.00 MB	N/A
Metaspace	NON_HEAP	98.53 MB	N/A	N/A	98.53 MB	N/A
Compressed Class Space	NON_HEAP	12.10 MB	1.00 GB	1.18%	12.10 MB	1.00 GB
G1 Eden Space	HEAP	22.00 MB	N/A	N/A	177.00 MB	N/A

Overview MBean Browser Triggers System Memory Threads Diagnostic Commands

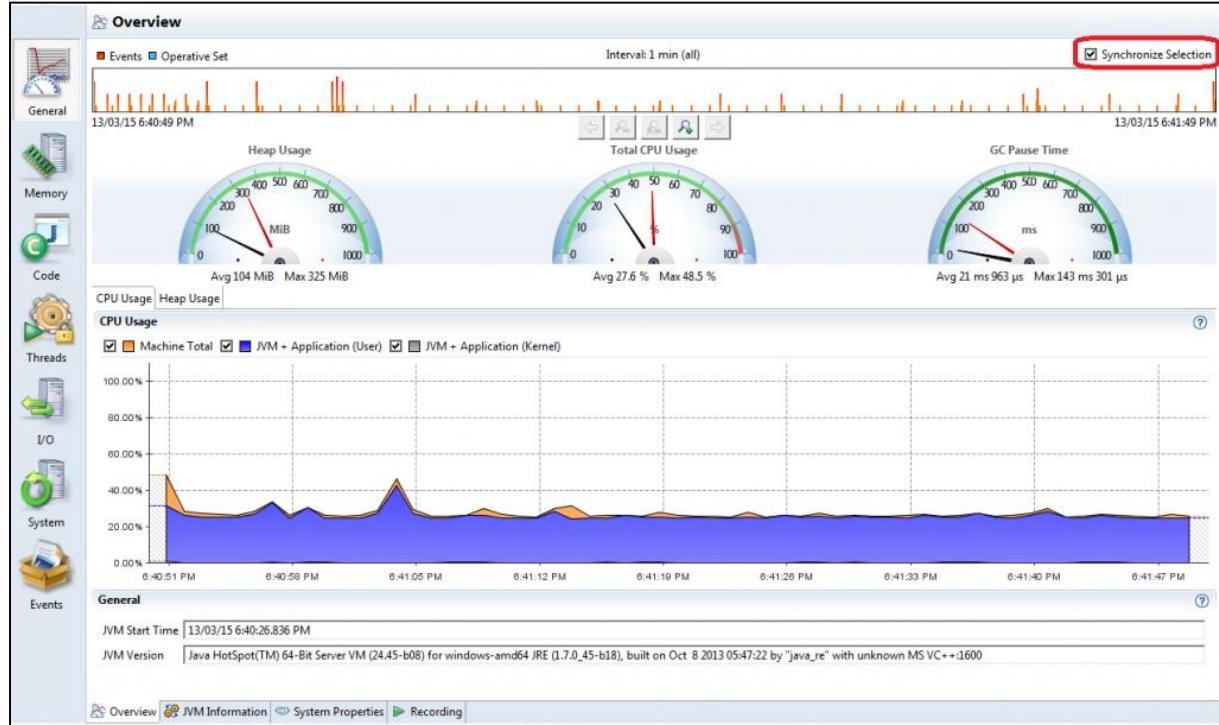
Java Mission Control

- Threads tab
- Thread state
- Lock name
- Deadlock
- Blocked count
- Per thread CPU usage
- Amount of memory allocated by a given thread since it was started



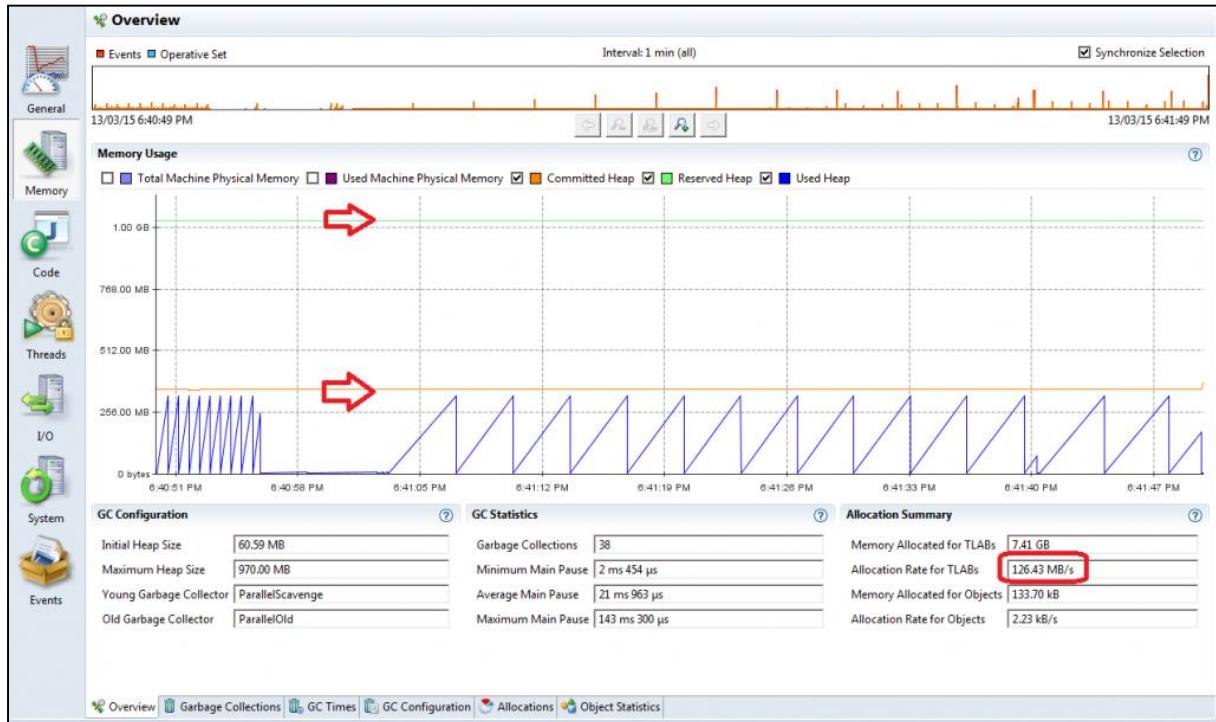
Java Mission Control

- Flight recorder - Initial screen
 - View JVM args
 - You may set
 - time range for analyzing



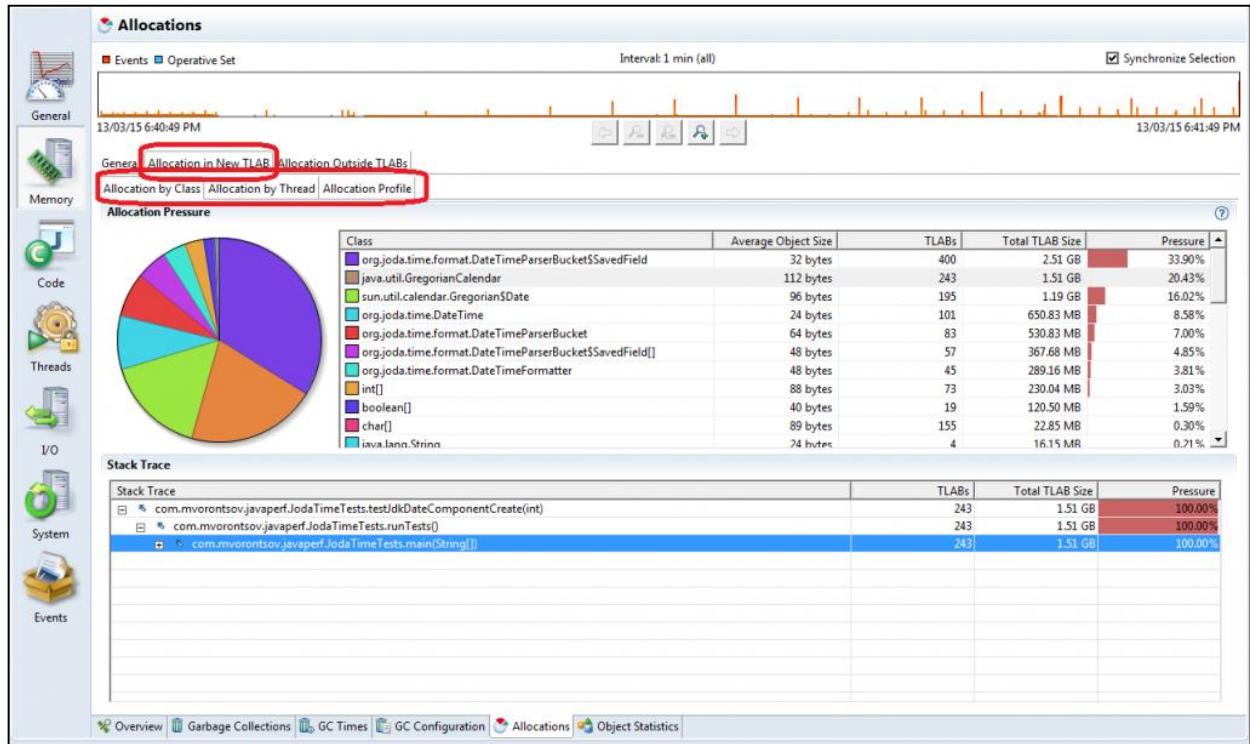
Java Mission Control

- Flight recorder memory tab
 - Machine RAM
 - Java heap usage
 - Garbage collections – when, why, for how long and how much space was cleaned up.
 - Memory allocation – inside / outside TLAB, by class/thread/stack trace.
 - Heap snapshot – number / amount of memory occupied by class name



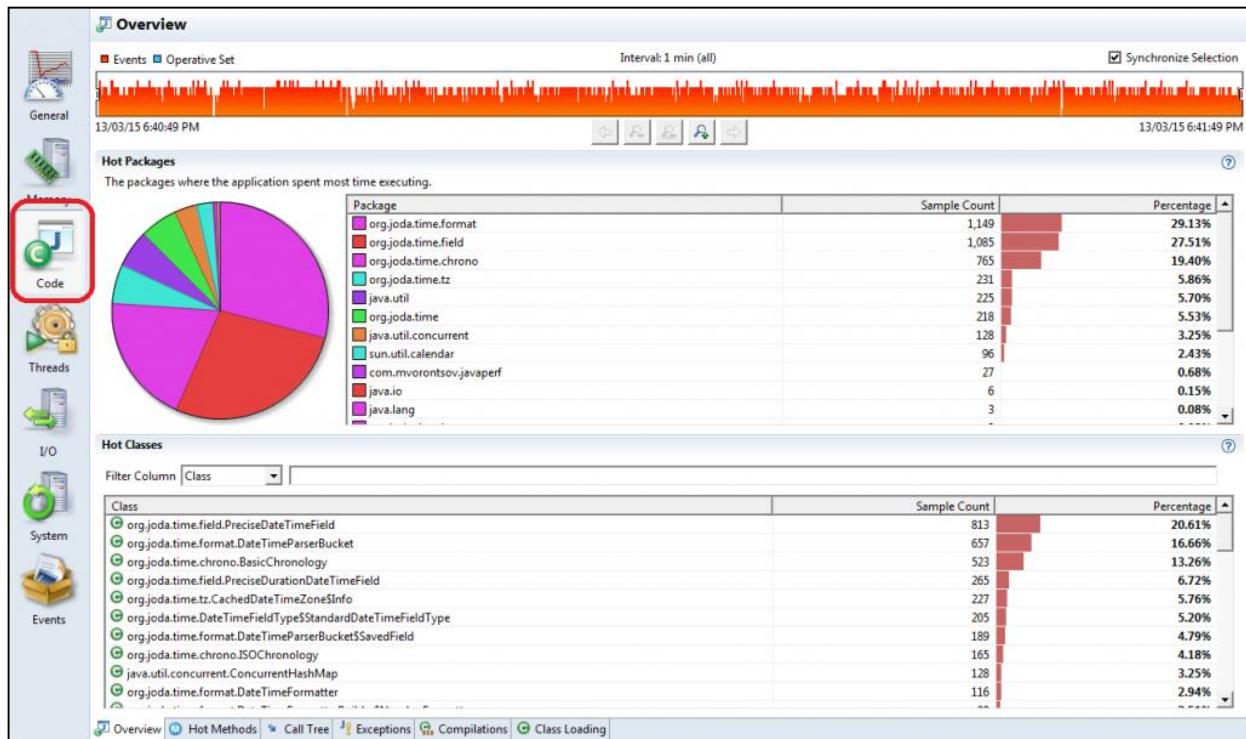
Java Mission Control

- Flight recorder allocation tab
 - Object allocations
 - By class
 - By thread



Java Mission Control

- Flight recorder code tab
 - Method hotspots
 - Packages hotspots
 - Useful for CPU optimization



Java Mission Control

- Other tabs
 - Exceptions
 - I/O
 - Threads

The screenshot shows the Java Mission Control interface with two main tabs highlighted: 'Exceptions' and 'Stack Trace'.
Exceptions Tab: This tab displays a table of exceptions. The columns are 'Class', 'Thread', and 'Message'. The data shows multiple instances of various exception classes, primarily from the java.lang and java.rmi packages, occurring on RMI TCP Connections. The 'Message' column contains details like 'no such object in table', 'serialPersistentFields', and stack traces.
Stack Trace Tab: This tab shows a detailed call stack for a specific thread. The stack trace starts with 'java.lang.Throwable.<init>(String)' and goes through several layers of Java and RMI classes, ending at 'java.lang.Thread.run()'. The right side of the table has a 'Count' column with all values set to 1, indicating each entry is unique.
Bottom Navigation: A navigation bar at the bottom includes links for Overview, Hot Methods, Call Tree, Exceptions (which is selected), Compilations, and Class Loading.



Java Concurrency



- Basics brief
- Executors
- Callable & Future
- Atomic Concurrent
- Lock API (Lock, Criteria)
- Barriers
- Fork-Join

Subclassing Thread and Overriding run

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try {  
                sleep((long)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! " + getName());  
    }  
}
```

Main method:

```
public class TwoThreadsTest {  
    public static void main (String[] args) {  
        new SimpleThread("Jamaica").start();  
        new SimpleThread("Fiji").start();  
    }  
}
```

Output:

```
0 Jamaica  
0 Fiji  
1 Fiji  
1 Jamaica  
2 Jamaica  
2 Fiji  
3 Fiji  
...  
6 Jamaica  
7 Jamaica  
7 Fiji  
8 Fiji  
9 Fiji  
8 Jamaica  
DONE! Fiji  
9 Jamaica  
DONE! Jamaica
```

Implementing Runnable

```
public class SimpleRunnable implements Runnable{  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + Thread.currentThread().getName());  
            try {  
                Thread.sleep((long)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! " + Thread.currentThread().getName());  
    }  
}
```

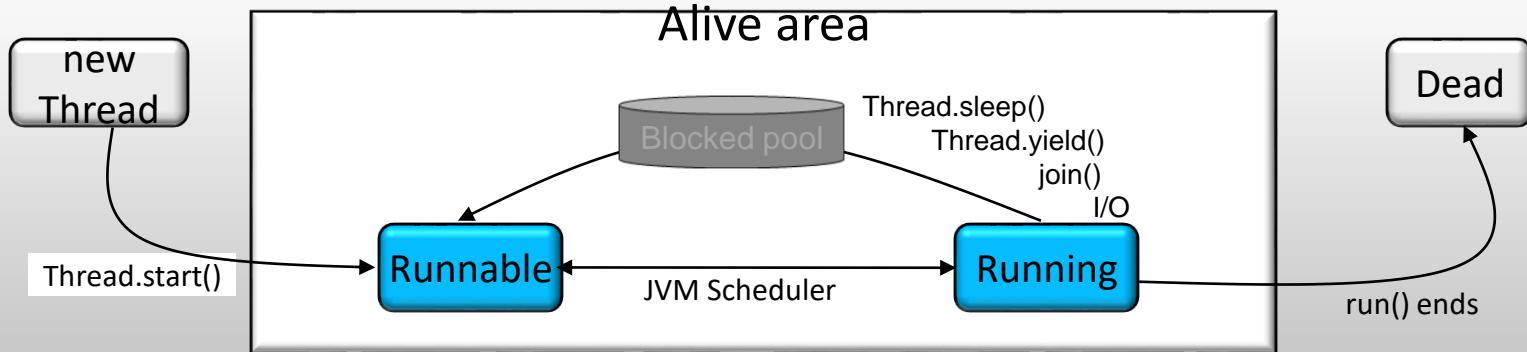
Main method:

```
public class TwoThreadsTest {  
    public static void main (String[] args) {  
        SimpleRunnable runner=new SimpleRunnable ();  
        Thread t1=new Thread(runner,"Jamaica");  
        Thread t2=new Thread(runner,"Fiji");  
        t1.start();  
        t2.start();  
    }  
}
```

Output:

```
0 Jamaica  
0 Fiji  
1 Fiji  
1 Jamaica  
2 Jamaica  
2 Fiji  
3 Fiji  
...  
6 Jamaica  
7 Jamaica  
7 Fiji  
8 Fiji  
9 Fiji  
8 Jamaica  
DONE! Fiji  
9 Jamaica  
DONE! Jamaica
```

The Life Cycle of a Thread



Basic states of the Thread:

- New Thread
- Running
- Not Runnable
- Dead

States of the Thread – Not Runnable

Basic control:

- Sleep – moves the thread to a non-Runnable state for a period of time (ms)
 - Usually the simplest way to delay threads or main
 - Note: blocks the thread at least to the specified time – not exactly
- Yield – moves the Running thread to the Runnable pool (Equals to sleep(0))
 - Usually for giving other low priority thread a chance to run
- Join – moves the running thread to a non-Runnable state until a specific thread ends
 - Delays the caller until the referenced thread ends
 - Is absolute – not like priority
- All methods throws InterruptedException
 - When thread are out of the blocking state before time
 - Might happen due to OS activity
- I/O Block – same occurs for connect(), read() & write()
- Blocked threads returns to runnable state
 - never to running (!)

```
try {  
    Thread.sleep(3000)  
} catch (InterruptedException e) {}
```

```
try {  
    Thread.yield()  
} catch (InterruptedException e) {}
```

```
...  
Thread t=new Thread(runner);  
t.start();  
try {  
    t.join()  
} catch (InterruptedException e) {}  
// all the work here happens after t ends  
...
```

States of the Thread – Dead

- the run method must terminate naturally
- stop method – *deprecated!!!*
 - This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack)

Daemon Threads

- Threads keep on running even after main thread ends
- Means that the VM still ‘on the air’ until the last thread dies
- In order to kill a thread when system exits it has to be a daemon
- Thread can be set to behave as daemon via `setDaemon(boolean)`
- Thread can be checked via `isDaemon()`

- Garbage collection is a daemon thread
 - therefore doesn’t last after system exit
 - That’s why sometimes object may never get the `finalize()` call



The isAlive Method

- Returns true if:
 - If the thread has been started and not stopped
 - the thread is Runnable or Not Runnable
- Returns false if:
 - the thread is New Thread or Dead

Synchronizing Threads

- Separate, concurrently running threads do share data and must consider the state and activities of other threads
- One such set of programming situations are known as **producer/consumer** scenarios where the producer generates a stream of data which then is consumed by a consumer
- The code segments within a program that access the same object from separate, concurrent threads are called critical sections
- In the Java language, a critical section can be a block or a method and are identified with the **synchronized** keyword
- The Java platform then associates a lock with every object that has synchronized code.

Locking an Object - Example

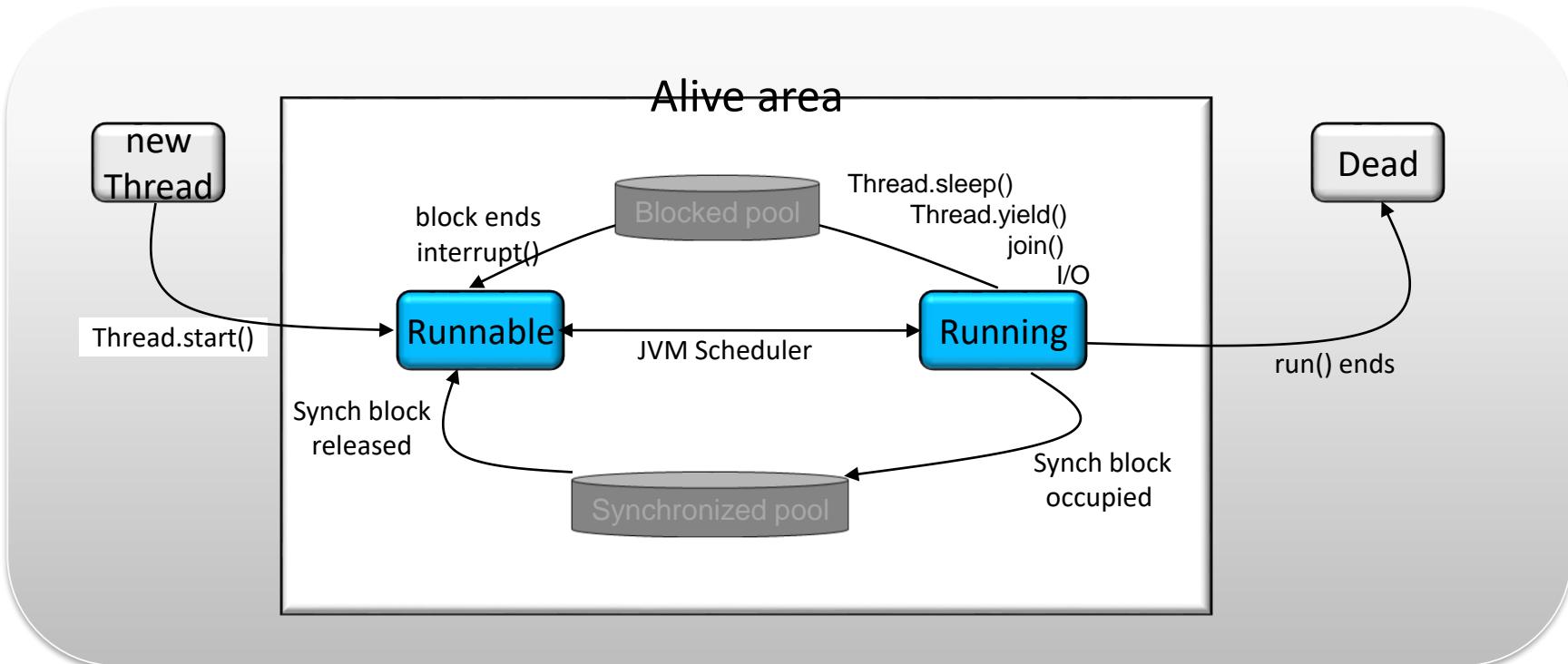
```
public class Car{  
    ...  
    public synchronized void drive(){  
        //this Car is locked by a Driver  
        ...  
        //this Car is unlocked by a Driver  
    }  
}
```

- Only one Driver can drive the Car at a time
- Only one thread at a time can own an object's monitor

```
public class Car{  
    ...  
    //It's not synchronized now  
    public void drive(){...}  
}  
  
public class Person{  
    public void use(Car c){  
        synchronized(c){  
            ...  
            c.drive();  
            ...  
        }  
    }  
}
```

- More than one Driver can drive the Car at a time

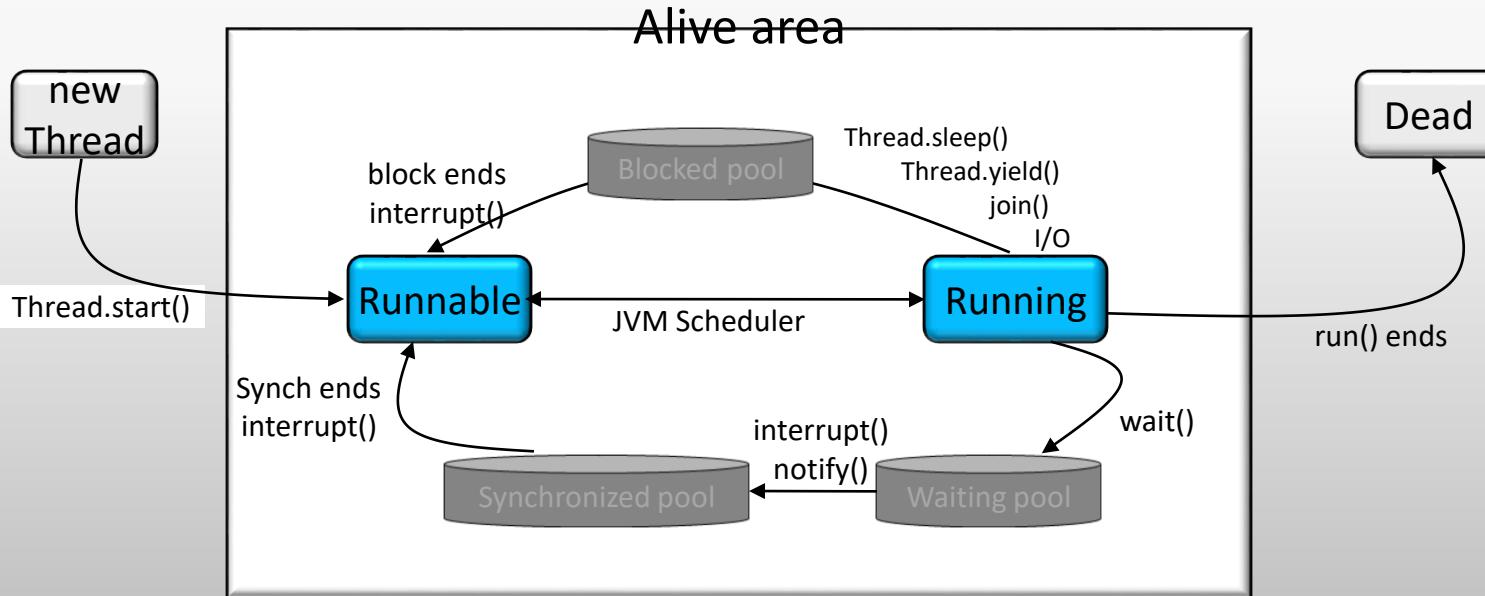
Synchronization



Using the notify and wait Methods

- *wait(), wait(long timeout)*
 - Causes current thread to wait until another thread invokes the *notify()* method or the *notifyAll()* method for this object, or a specified amount of time has elapsed.
- *notify()*
 - Wakes up a single thread that is waiting on this object's monitor
- *notifyAll()*
 - Wakes up all threads that are waiting on this object's monitor

Using the notify and wait Methods



Precondition to wait and notify Methods

- The current thread must own this object's monitor
- A thread becomes the owner of the object's monitor in one of three ways:
 - By executing a synchronized instance method
 - By executing the body of a synchronized
 - For objects of type Class, by executing a synchronized static method
- Only one thread at a time can own an object's monitor

Reentrant Locks

- The Java runtime system allows a thread to re-acquire a lock that it already holds because Java locks are reentrant

```
public class Reentrant {  
    public synchronized void a() {  
        b();  
        System.out.println("here I am, in a()");  
    }  
    public synchronized void b() {  
        System.out.println("here I am, in b()");  
    }  
}
```

```
MyThread t=new MyThread( new Reentrant());  
t.start(); //run() method calls Reentrant.a()
```

Output:

here I am, in b()
here I am, in a()

The method interrupt

- Interrupts the thread
 - If the current thread is interrupted by another thread while it is waiting, then an *InterruptedException* is thrown
 - This exception is not thrown until the lock status of
 - this object has been restored as described above
 - This method should only be called by a thread that is the owner of this object's monitor

Thread Local

- Keeps state for individual threads
- Each thread has its own independently initialized copy of the variable
- Differs from local variables
 - Managed in the language level rather than stack management (makes thread stack lighter)
 - Obtain a direct access to data using the thread as a hash-map key
- Used by threads to store / load session states between calls
- *java.lang.ThreadLocal*
- Data is simply accessible via get()/set() methods
- Thread entry is automatically discarded when a thread dies

Thread Local

Example:

```
public class MyThread extends Thread
{
    // static Hashtable stuff;
    public static ThreadLocal stuff = new ThreadLocal();

    public void run() {

        // each thread must have a different hashtable
        Hashtable h= new Hashtable();

        //load session data on hashtable
        ...
        //assign it to ThreadLocal
        stuff.set(h);
        handleData();
    }

    public void handleData(){
        if (((Hashtable)stuff.get()).containsValue("Foo"))
            doSomething(); // run code which can refer to hash table
    }
}
```

Thread Executor

An implementation of *java.util.concurrent.Executor*

- Thread Pool Executor
 - A thread pool service
 - Offers powerful control over the pool
 - Can expand and shrink the pool according to call intensity
- Better since
 - Manages shared resources among threads
 - Offers monitoring and reporting [for example: no' of completed tasks]
 - Supports Callable objects execution (not just Runnable) - later
 - Manages a fully featured queue
 - Min/max sizes
 - On-demand service
 - Keep-alive time
 - Can be shut down and reject calls
 - Offers scheduling assignments - better & lighter than Timers

Executor queue types

- Synchronous queue
 - means that a call is handled rapidly – if no available thread – a new one is constructed
 - max size is irrelevant and ignored
- Blocking queue
 - means that threads are created until the queue is full (at max size)
 - then – queuing begins
- All supports both *Runnable* & *Callable*



Callable

- Callable is an interface with a single method
 - *Callable<T> implementers must provide public <T> call() throws Exception*
- Similar to Runnable – *run()* in that they are both executed by threads
- But differs with the following capabilities:
 - Returns a result of the execution
 - Throws checked exceptions
 - Several Callable objects can be executed with a single call to the Executor service
 - One thread can invoke several Callable objects during its life time

Callable

Example:

```
import java.util.concurrent.Callable;

public class ClientListener implements Callable<Request> {
    private ClientInputStream in=null;

    public ClientListener (ClientInputStream in){
        this.in=in;
    }

    public Request call() throws ConnectionException {
        try {
            return in.acceptRequest();
        }catch(Exception ex){
            throw new ConnectionException ("Error processing client request");
        }
    }
}
```

Void type safe Callable Example:

```
import java.util.concurrent.Callable;

public class VoidCallable implements Callable<Void> {

    public Void call() throws Exception {
        //do job
        //Void type-safe return value must be NULL
        return null;
    }
}
```

Executors Utility Class

Creates the following:

- *FixedThreadPool*
 - A fixed thread pool that will use only the given amount of threads to execute *Runnable* & *Callable* objects
 - If all threads are busy, *Runnable* objects that are not currently running – blocks
 - Asynchronous queue
- *ScheduledThreadPool*
 - Offers the ability of scheduling *Runnable* & *Callable* objects execution

Executors Utility Class

Creates the following:

- *CachedThreadPool*
 - Synchronous queue
 - Creates new threads every time a new execution is required and all threads are busy
 - Therefore
 - Non blocking queue
 - Max threads in the pool is irrelevant
 - Threads are removed automatically after waiting 60 sec for a task
- *SingleThreadExecutor*
 - Handles all registered *Runnable* and *Callable* objects via a single thread

Executors Utility Class

- All available through static methods
- Most takes fixed or minimum pool size as *int*
- Returns an *Executor* implementation [*ExecutorService*]
- Threads are created according to
 - Queue type and policy
 - Queue thread factory implementation [later]
- Each thread may invoke more than one Callable object
 - For example: when a fixed queue with the size of 2 is invoking 4 callable objects, 2 threads will be created and used to invoke all 4

Executors Utility Class

ExecutorService - execution methods

- *Future<Void> submit(Runnable)*
 - Returns Future with null result
 - Doesn't throws any exception
 - Is a non-blocking operation

- *Future<T> submit(Callable<T>)*
 - Returns Future<T>
 - Is a non-blocking operation

Executors Utility Class

ExecutorService - execution methods

- `void execute(Runnable)`
 - Specified in `java.util.concurrent.Executor`
 - Returns void
 - Doesn't throws any exception
 - Is a non-blocking operation
- `List<Future<T>> invokeAll(Collection<Callable<T>>)`
 - Specified in `java.util.concurrent.ExecutorService`
 - Returns and array of `Future<type of the call() method result>`
 - Throws `InterruptedException`
 - Is a blocking operation
- `<T> invokeAny(Collection<Callable<T>>)`
 - Specified in `java.util.concurrent.ExecutorService`
 - Returns a single value [`type of the call() method result`] of the task that has completed
 - Other uncompleted tasks are canceled
 - Throws `InterruptedException`
 - Is a blocking operation

Thread Executor

Runnable Example:

```
public class Worker implements Runnable {  
    public void run() {  
        System.out.println("New Worker");  
    }  
  
public class User{  
    public static void main(String args[]) {  
        Executor e = Executors.newFixedThreadPool(2);  
        e.execute(new Worker());  
        e.execute(new Worker());  
        e.execute(new Worker()); //probably will wait  
    }  
}
```

Thread Executor

Callable Example:

```
ExecutorService executor=Executors.newCachedThreadPool();
Callable<Request> client1=new ClientListener<Request>(new ClientInputStream (...));
Callable<Request> client2=new ClientListener<Request>(new ClientInputStream (...));
Callable<Request> client3=new ClientListener<Request>(new ClientInputStream (...));

//create a callable collection in order to execute all at once
Collection<Callable<Request>> col=new HashSet<Callable<Request>>();
col.add(client1);
col.add(client2);
col.add(client3);

try {
    List<Future<Request>> requests = executor.invokeAll(col);
    ...
} catch (InterruptedException e) { ... }
```

- Completed workers are those who
- Executed call() and returned T as a result
 - Failed with exception – result is null
 - Cancelled
 - invokeAny & invokeAll catches call() exceptions but doesn't delegate them



Future

Future<T>

- A result wrapper
- Is returned from both blocking & non-blocking execution
- For uncompleted execution –
 - *Future.get()* blocks until completed
 - Execution can be investigated via *Future.isDone()*, *Future.isCancelled()*
 - Execution can be canceled via *Future.cancel()*
- For completed execution –
 - *Future.get()* returns immediately

Future<T>

- *Get()* returns
 - T – when execution successfully completed
 - If the execution is based on *Callable<T>* - it might end with T or exception
 - If the execution is based on Runnable – it always returns null (*Future<Void>*)
- Execution Exception
 - *Get()* throws checked Exception
 - In case of Callable throwing exception – it will be wrapped by Future exception
 - Use *e.getCause()* in order to get the actual execution exception

Getting *Future* with *Runnable* Example:

```
public class Worker implements Runnable {  
    public void run() {  
        System.out.println("New Worker");  
    }  
}  
  
public class User{  
    public static void main(String args[]) {  
        Executor e = Executors.newFixedThreadPool(2);  
        Future<Void> f = e.submit(new MyRunnable());  
        f.get();  
        //we get here only after the task is completed...  
    }  
}
```

Getting Future with *Callable* Example:

```
ExecutorService executor=Executors.newCachedThreadPool();
Callable<Request> client1=new ClientListener<Request>(new ClientInputStream (...));
Callable<Request> client2=new ClientListener<Request>(new ClientInputStream (...));
Callable<Request> client3=new ClientListener<Request>(new ClientInputStream (...));

//create a callable collection in oprder to execute all at once
Collection<Callable<Request>> col=new HashSet<Callable<Request>>();
col.add(client1);
col.add(client2);
col.add(client3);

try {
    List<Future<Request>> requests = executor.invokeAll(col);
    for(Future<Request> f: requests){
        try{
            Request curr=f.get();
        }catch(Exception e){
            Exception origin = e.getCause();
        }
    }
    ...
} catch (InterruptedException e) { ... }
```

Executors Thread Factory

Executors can be created with

- Default thread factory
 - Simply generates threads when asked to
 - Queue type is the one that responsible for using the factory according to its policy
- Custom thread factory
 - Is developed as a separate class and implements *java.util.concurrent.ThreadFactory*
 - Is assigned to the executor via Executors static methods
 - May be used for extra logic when creating threads used by Executors like:
 - Defining thread pool size
 - Set threads with customized names & priorities
 - Provide daemon threads

```
public class SimplestThreadFactory implements ThreadFactory{  
  
    public Thread newThread(Runnable r) {  
        return new Thread(r);  
    }  
}
```

Executors Thread Factory

Example:

```
public class CustomThreadFactory implements ThreadFactory{  
  
    private int priority;  
    private boolean daemon;  
  
    public CustomThreadFactory (int priority, boolean daemon){  
        this.priority=priority;  
        this.daemon=daemon;  
    }  
  
    public Thread newThread(Runnable r) {  
        Thread newThread=new Thread(Runnable);  
        newTread.setPriority(priority);  
        newThread.setDaemon(daemon);  
        return newThread;  
    }  
}
```

Usage:

```
CustomThreadFactory factory = new CustomThreadFactory (5,true);  
ExecutorService executor=Executors.newCachedThreadPool(factory);
```

Fork-Join

- Lightweight processing with *ForkJoinTask<V>* & Pool
- Some terms:
 - Heavyweight processing = processing in separate stack
 - Threads helps in that
 - Stack values has to be re-assigned every time thread uses CPU time
 - Synchronization might be needed
 - Known as ‘fork’
 - Lightweight processing = processing in the same stack
 - Linear processing
 - Processing time might be short, long or very long..
 - For short time – we can use the same thread or new thread
 - For long time – we can use new thread

Fork-Join

For very long tasks we might want to:

- Split task into small independent parts
- Fork each part to solve it separately
 - In the same thread pool dedicated to that long task
- Join all parts
- Compose the result out of them

Fork-Join

Java 7 offers:

- ForkJoinTask – lightweight task (like Callable-Future)
 - *RecursiveAction* – a task that doesn't return a result
 - *RecursiveTask<E>* - a task that results in E type
 - Both has *invokeAll(..)* method to fork other subtasks
- *ForkJoinPool* – An *ExecutorService* implementation
 - Designed for forking tasks and its subtasks
 - Takes number of processors to its constructor
 - Default constructor is set according to the *Runtime.availableProcessors()*

Fork-Join

- Example:

```
public class ElementTask extends RecursiveAction{
    private Node n;
    public ElementTask(Node n) {this.n=n;}
    @Override
    public void compute(){
        if(node.getNodeValue().equals("leaf")){
            ..... Process node data
        }else { // node is composite of sub-leafs.....
            NodeList subs=n.getChildNodes();
            Collection<ElementTask> subTasks=new HashSet<>()
            for(int i=0;i<subs.getLength();i++){
                subTasks.add(new ElementTask(subs.item(i)));
            }
            invokeAll(subTasks); //registers more tasks to the pool..
        }
    }
}
```

```
ElementTask t=new ElementTask(xmlDocumentNode);
ForkJoinPool pool=new ForkJoinPool ();
pool.invoke(t);
```



CompletableFuture

- Since Java 8
- *CompletionStage<T>* interface
 - Used for breaking tasks into conditional continues sub-tasks
- *CompletableFuture*
 - Implements *CompletionStage*
 - Provides operations for creating sub-tasks pipelines
 - Supports LAMBDA exp.

CompletableFuture

- Provides operations like:

- *thenAccept(Consumer<? super T> action)*
 - *supplyAsync(Supplier<U> supplier)*
 - *thenApply(Function<? super T, ? extends U> fn)*
 - *thenRun(Runnable action)*
-
- All result with *CompletionStage<T>* - so pipelines may be easily coded
 - All may use current thread or obtain different one from given executor - *asyncXXX()*
 - All may be executed on different executors than *ForkJoinPool..commonPool()*

CompletableFuture

- Examples:

Method ref pipeline:

```
CompletableFuture.supplyAsync(me::findRemoteControl)
    .thenApply(kodi::searchMovie)
    .thenAccept(me::watchMovie);
```

LMBDA Exp. Pipeline:

```
CompletableFuture.supplyAsync(()->me.findRemoteControl("under the sofa"))
    .thenApply((rc)-> kodi.favorateChannel(rc,"Pulp Fiction")
    .thenAccept((mov)->me.watchMovie(mov));
```

- Methods relates to other *CompletableFuture<T>* instances:

- acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)
- applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn)
- runAfterBoth(CompletionStage<?> other, Runnable action)
- runAfterEither(CompletionStage<?> other, Runnable action)



Concurrent Atomic

- *java.util.concurrent.atomic* package
 - Provides thread-safe wrappers of atomic data
 - Wait, isn't it 'volatile' keyword responsibility?

Concurrent Atomic

Volatile

- Synchronizes access to primitive allocations
- Should be used when variables are about to be shared and modified by multiple threads
- Volatile compared with synchronization:
 - May ‘synchronize’ null values
 - Can be applied to primitives as well
 - Well supported since Java 5 (don’t use it in older versions)
 - Non-blocking (just for the single action itself...)

Concurrent Atomic

So, why do we need Atomic locks if we can use volatile?

- volatile cannot block, means we cannot manipulate a value and read it in an atomic context
- Volatile doesn't solve this:
- Volatile doesn't support conditional updates...

```
int x=100;  
...  
if(x==200){  
    x=x*10;  
    x++;  
}
```

Concurrent Atomic

Concurrent Atomic wrappers

- Provides volatile access to its value
- Extends it with general methods
 - `set()`, `get()`
 - `compareAndSet(expectedValue, updatedValue)`
- Extends it with type specific methods
 - For `int`: `addAndGet(int delta)`, `decrementAndGet()` ...

Concurrent Atomic

A little about *compareAndSet(expected, updated)*

- Performs volatile read on wrapped value
- If expected == value
 - Volatile updates the value with updated
 - Return true
- If expected != value
 - Return false
- Threads might retry to change data on ‘false’ result
 - But are not blocked due to this method invocation

Concurrent Atomic

List of atomic wrappers:

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicLong
- AtomicLongArray
- AtomicReference
- AtomicReferenceArray

Concurrent Atomic

Example:

```
private AtomicInteger a=new AtomicInteger(100);
..
public int getValue(){
    return a.get();
}

public boolean update (int current, int value){
    return a.compareAndSet(current,value);
}

public int increment(){
    return a.incrementAndGet();
}
```

Concurrent Locking

java.util.concurrent.locks

- Provides framework for conditional locking
- Much flexible than the classical build-in mechanism
- Lock – defines the lock policy
- Condition – manipulate executions on a Lock
- Read-Write lock – combination of 2 locks
 - One for reading and one for writing

Concurrent Locking

Lock interface

- *lock()* - acquires a lock
- *unlock()* – releases it
- *lockInterruptibly()* – locks until thread gets interrupted
- *tryLock()* – locks only if it is free on execution
- *tryLock(long time, TimeUnit unit)* – waits for it to be free and lock
- *newCondition()* - returns a condition bound to the lock

Concurrent Locking

Condition interface

- Offers enhanced Lock control
- Multiple conditions can be obtained and managed on a Lock
- Methods:
 - *signal()* – wakes up waiting thread
 - *signalAll()* – wakes up all waiting threads on that Lock
 - *await()* – causes calling thread to wait until signaled or interrupted
 - *await(long time, TimeUnit unit)*, *await(long nanos)*

Concurrent Locking

Concrete Locks

- *ReentrantLock*
 - Acts like classic thread synchronization
 - Thread can lock only if not locked already
 - Many Conditions can be created
 - Provides informative methods
 - *isLocked()*
 - *getLockQueueLength()*

Concurrent Locking

Concrete Locks

- *ReentrantReadWriteLock*
 - Maintains a un-fair read-write time slicing – can be set to fair
 - Only one thread can write at a time
 - Several reader threads can read at the same time
 - *readLock()* obtains the Lock for reading
 - *writeLock()* obtains the lock for writing

Concurrent Locking

Simple Example:

```
private ReentrantLock lock=new ReentrantLock();
..
public void method(){
    lock.lock();
    // do some thread sensitive tasks
    lock.unlock();
}
```

Concurrent Locking

Concrete Locks

- CyclicBarrier
 - Blocks specified number of threads – then releases all
 - Barrier is created with a given counter
 - Threads uses barrier *await()* method to block
 - When the number of waiting threads reached the counter – all waiting threads are released
 - Cyclic means - A reusable threads barrier

Concurrent Locking

Simple Example - CyclicBarrier:

```
public class Task implements Runnable{  
    private CyclicBarrier barrier;  
  
    public Task(CyclicBarrier barrier) {  
        this.barrier = barrier;  
    }  
  
    @Override  
    public void run(){  
        try {  
            System.out.println(Thread.currentThread().getName() + " is waiting on barrier");  
            barrier.await();  
            System.out.println(Thread.currentThread().getName() + " has crossed the barrier");  
        } catch (InterruptedException ex) {  
            ...  
        }  
    }  
}
```

Concurrent Locking

Simple Example - CyclicBarrier:

```
public static void main(String args[]) {  
  
    final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){  
        @Override  
        public void run(){  
            //This task will be executed once all thread reaches barrier  
            System.out.println("All parties are arrived at barrier, lets play");  
        }  
    });  
  
    Thread t1 = new Thread(new Task(cb), "Thread 1");  
    Thread t2 = new Thread(new Task(cb), "Thread 2");  
    Thread t3 = new Thread(new Task(cb), "Thread 3");  
  
    t1.start();  
    t2.start();  
    t3.start();  
}
```

Output:

```
Thread 1 is waiting on barrier  
Thread 3 is waiting on barrier  
Thread 2 is waiting on barrier  
All parties are arrived at barrier, lets play  
Thread 3 has crossed the barrier  
Thread 1 has crossed the barrier  
Thread 2 has crossed the barrier
```

Concurrent Locking

Concrete Locks

- *CountDownLatch*
 - Blocks threads – when counter reached zero (0) – releases all threads
 - *CountDownLatch* is created with a given counter
 - Threads uses *await()* method to block
 - *countDown()* method decrements the counter – and if the counter reaches zero (0) – all waiting threads are released
 - Unlike *CyclicBarrier*, cannot be reset and reused

Concurrent Locking

Simple Example - CountDownLatch:

```
public class Task implements Runnable{  
    private CountDownLatch cdl;  
  
    public Task(CountDownLatch cdl) {  
        this.cdl = cdl;  
    }  
  
    @Override  
    public void run(){  
        try {  
            System.out.println(Thread.currentThread().getName() + " is waiting on count down latch");  
            //here – part of the thread job – is to update the counter when finished  
            cdl.countDown();  
            cdl.await();  
            System.out.println(Thread.currentThread().getName() + " has crossed the count down latch");  
        } catch (InterruptedException ex) {  
            ...  
        }  
    }  
}
```

Concurrent Locking

Concrete Locks

- Semaphore
 - Controls threads access to limited resources
 - Semaphore is created with a given ‘fair’ flag
 - Fairness is trying to manage first-in-first-out policy on blocked threads
 - Threads uses *tryAcquire(maxWait, TimeUnit)* to lock
 - *release()* / *release(numOfThreadsToPermit)* – releases the lock

Concurrent Locking

Simple Example – Semaphore & resource pool:

```
public class ResourcePool<T> {  
  
    private final Semaphore sem = new Semaphore(MAX_RESOURCES, true);  
    private final Queue<T> resources = new ConcurrentLinkedQueue<T>();  
  
    public T getResource(long maxWaitMillis) throws InterruptedException, ResourceCreationException {  
  
        // First, get permission to take or create a resource  
        if( sem.tryAcquire(maxWaitMillis, TimeUnit.MILLISECONDS); ){  
  
            // Then, actually take one if available...  
            T res = resources.poll();  
            if (res != null) return res;  
  
            // ...or create one if none available  
            try {  
                return createResource();  
            } catch (Exception e) {  
  
                // release if we failed to create a resource!  
                sem.release();  
                throw new ResourceCreationException(e);  
            }  
        }  
    }  
  
    public void returnResource(T res) {  
        resources.add(res);  
        sem.release();  
    }  
}
```

Concurrent Locking

Reentrant with Conditions Example:

```
public class BoundedBuffer {  
    final Lock lock = new ReentrantLock();  
    final Condition notFull = lock.newCondition();      //if not full - execute  
    final Condition notEmpty = lock.newCondition();     //if not empty - execute  
    final ArrayList items = new ArrayList(100);  
  
    public void put(Object x) throws InterruptedException {  
        lock.lock();  
        try {  
            while (items.size()==100)  
                notFull.await();  
            items.add(x);  
            notEmpty.signal();  
        } finally { lock.unlock(); }  
    }  
    public Object pop() throws InterruptedException {  
        lock.lock();  
        try {  
            while (item.size()== 0)  
                notEmpty.await();  
            Object x = items.remove(items.size()-1);  
            notFull.signal();  
            return x; }  
        finally { lock.unlock(); }  
    }  
}
```

Concurrent Locking

Java 8 - Optimistic reading

- When a reader obtains a non-exclusive lock on a resource it receives a stamp
- On any writer update to the resource – the stamp gets updated
- Readers may use their stamps in order to validate the resource
 - If the stamp is the same as the one in the source – no writers obtained any lock
- Writers always update the resource stamp on completion
 - Stamps are much like ‘version’ in Hibernate/JPA

StampedLock class

- *tryOptimisticRead()*
 - used by readers in order to manage optimistic read locks on a resource
 - results with a non-zero stamp or zero if exclusively locked
- *validate(stamp)*
 - Returns true if resource has the same stamp (means no other writer flushed data)
- *unlockRead(stamp)* – releases the reader non-exclusive lock if stamp matches
- *unlockWrite(stamp)* - releases the writer exclusive lock if stamp matches



Concurrent Collections

- We got:
 - Old collections which are synchronized
 - *Collections.synchronizedXXX(...)*
- So what's the point?

Concurrent Collections

- Basic synchronization offers poor locking logic
 - No reading/writing considerations in Lists & Sets....
 - Maps locks – no matter which key is used....
- Instead of using ‘synchronized’ we can:
 - Use *ReadWriteLock* for Lists & Sets
 - Manage Lock instances according to keys for Maps
 - Or simply use the concurrent collections Java provides

Concurrent Collections

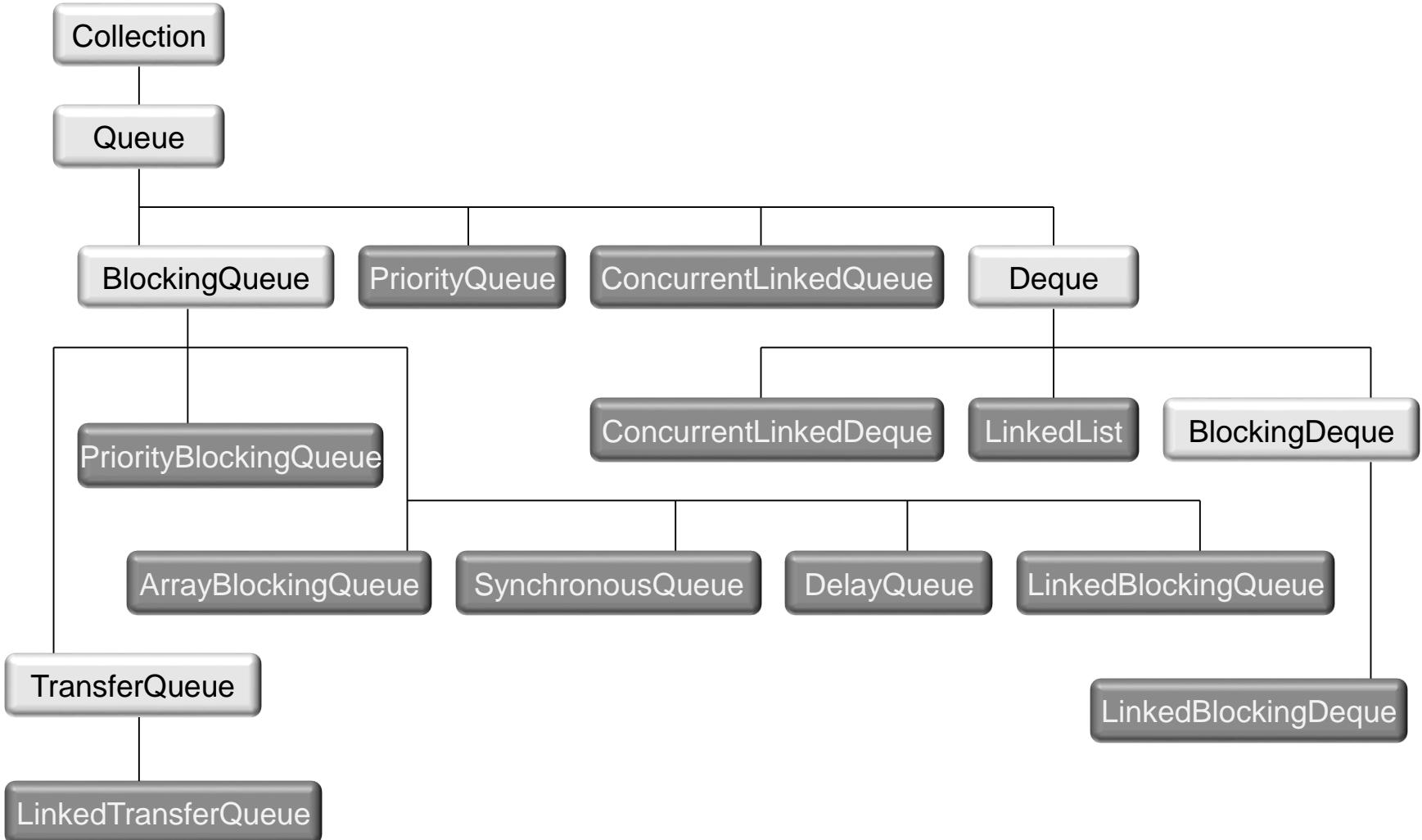
Working with concurrent collections

- *ConcurrentModificationException* are never thrown
- So, how does it work exactly ??
- Operations are done in a ‘dirty’ manner
- When using `clear()` while other threads produces values
 - you might not remove all objects
- When using `putAll()/addAll()` while other threads consumes values
 - you might end up with some missing objects
- `size()` might be inaccurate

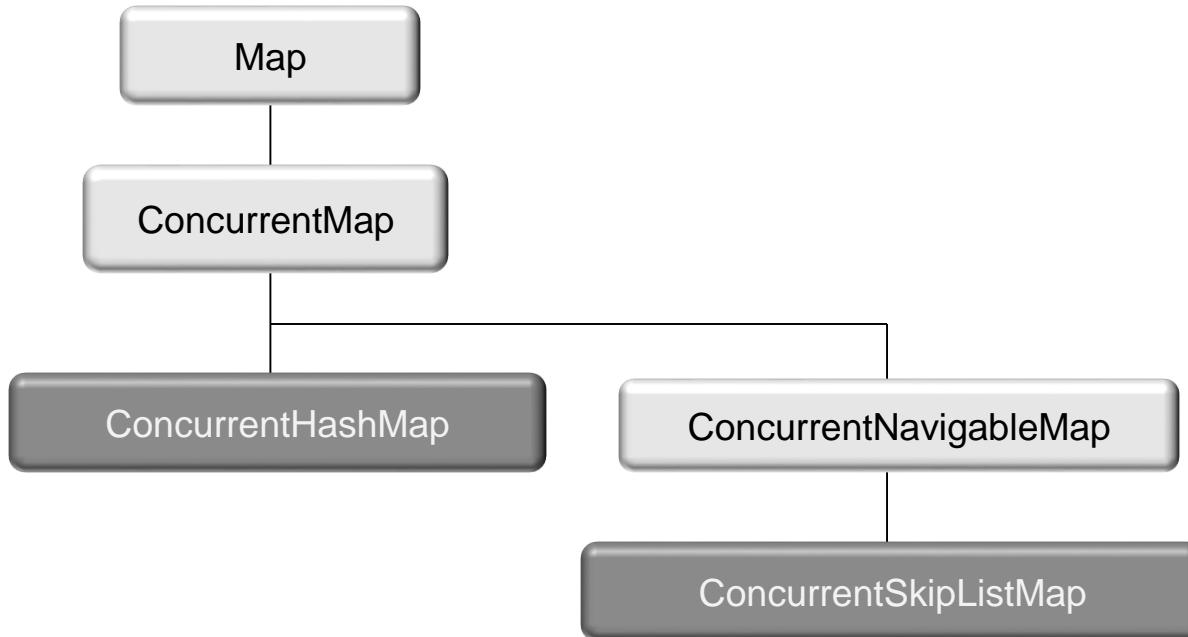
Concurrent Collections

- Java concurrent collection classification:
 - Queue
 - *put()*, *offer()* to insert
 - *poll()*, *remove()* to delete
 - *peek()* – to view
 - Map
- Super interfaces are:
 - *BlockingQueue*
 - *ConcurrentMap*

Concurrent Collections



Concurrent Collections



Concurrent Collections

BlockingQueue

- Null values are not permitted
- Manages remaining capacity
 - Might be bounded. If not, always shows *Integer.MAX_VALUE*
 - *take()* blocks when empty
 - *put()* blocks when remaining capacity = 0
- Atomic operations are thread-safe

Concurrent Collections

BlockingQueue – cont.

- Non-blocking methods
 - *add()* return true on success and *IllegalStateException* if full
 - *offer()* return true on success and false if full
 - *poll()* retrieves element or null if queue is empty
- Bulk operations (*addAll()*, *containsAll()*...)
 - Might partially execute and fail with exception

Concurrent Collections

ConcurrentMap

- Offers atomic locks
 - Key-based locks for *put()* & *get()*
- Additional methods
 - *putIfAbsent(key, val)* – assigns val if key has null value
 - *remove(key, val)* – removes entry if V equals current value
 - *replace(key, val)* – replaces current value with val
 - *replace(key, val, nv)* – assigns nv if current value equals val

Concurrent Collections

Concrete Blocking Queues

- *ArrayBlockingQueue*
 - Based on Java Array
 - Once created, cannot be resized
 - Constructor accepts capacity & ‘fair’ flag
 - true = consumer & producer thread goes through FIFO queue
 - false = no thread queue – might lead to threads starvation
- *SynchronousQueue*
 - Blocking queue with fixed zero size
 - Consumer *poll()* returns when producer *offer()* is executed

Concurrent Collections

Concrete Blocking Queues

- *PriorityBlockingQueue*
 - Sorted blocking queue
 - Uses natural ordering (*Comparable<T>*)
- *LinkedBlockingQueue*
 - Same functionality as *ArrayBlockingQueue* – but,
 - Backed by linked list (better for adding and removing elements)
 - Capacity can be specified

Concurrent Collections

Concrete Blocking Queues

- *DelayQueue*
 - Unbounded blocking queue
 - *offer(e, timeout, timeUnit)* – adds elements
 - *poll()* – results with null, if no element expired yet
 - Elements are organized according to their expiration

Concurrent Collections

Concrete Blocking Queues

- *TransferQueue* interface
 - Offered in JSE 7 release
 - Extends *BlockingQueue*
 - Allows a producer to use the queue in 2 methods
 - *put()* – as always, blocks when full or adds and returns
 - *transfer()* – which blocks until consumers uses *take()* or *poll()*
- *LinkedTransferQueue*
 - *TransferQueue* linked-list based implementation

Concurrent Collections

Concrete Blocking Queues

- BlockingDeque interface
 - Double-ended blocking queue
 - Concrete class: LinkedBlockingDeque

First Element (Head)

	<i>Throws exception</i>	true/false	<i>Blocks</i>	<i>Times out</i>
Insert	<u>addFirst(e)</u>	<u>offerFirst(e)</u>	<u>putFirst(e)</u>	<u>offerFirst(e, time, unit)</u>
Remove	<u>removeFirst()</u>	<u>pollFirst()</u>	<u>takeFirst()</u>	<u>pollFirst(time, unit)</u>
Examine	<u>getFirst()</u>	<u>peekFirst()</u>	<i>not applicable</i>	<i>not applicable</i>

Last Element (Tail)

	<i>Throws exception</i>	true/false	<i>Blocks</i>	<i>Times out</i>
Insert	<u>addLast(e)</u>	<u>offerLast(e)</u>	<u>putLast(e)</u>	<u>offerLast(e, time, unit)</u>
Remove	<u>removeLast()</u>	<u>pollLast()</u>	<u>takeLast()</u>	<u>pollLast(time, unit)</u>
Examine	<u>getLast()</u>	<u>peekLast()</u>	<i>not applicable</i>	<i>not applicable</i>

Concurrent Collections

Concrete Concurrent Maps

- *ConcurrentHashMap*
 - Acts like hash-map in the sense of maintaining pairs
 - *get()* (read) doesn't block
 - *put()* & *set()* uses atomic locks
 - Basically the whole table never gets synchronized

Concurrent Collections

Concrete Concurrent Maps

- *ConcurrentHashMap* – cont.
 - Constructor parameters: (IMPORTANT!)
 - - initial capacity (default: 16)
 - - load factor –
 - Close to zero means no collisions even with weak hash function
 - But will require lot of memory for hash values
 - Leave at default value - 0.75 - reasonable for java hash function
 - - concurrency level –
 - Sets the number of producer queues for this Map
 - Default is 16 (as default initial size)
 - Set according to the expected number of writer / producer threads
 - Low number might lead to writer / producer contentions

Concurrent Collections

Concrete Concurrent Maps

- *ConcurrentNavigableMap* interface
 - Base interface for concurrent sorted map, extends *SortedMap<K,V>*
 - Provides enhanced map functionalities
 - *ceilingKey(K)* – return the next key greater than or equals to K
 - *floorKey(K)* - return the next key less than or equals to K
 - *lowerKey(K)*, *higherKey(K)*
 - *lastEntry()*, *firstEntry()*
 - *pollLastEntry()*, *pollFirstEntry()* - removes

Concurrent Collections

Concrete Concurrent Maps

- *ConcurrentNavigableMap* interface – cont.
 - Offers some views on the Map table
 - *headMap(toKey)* – returns sub-map with all keys less than toKey
 - *tailMap(fromKey)* - returns sub-map with all keys greater than fromKey
 - *subMap(fromKey, toKey)*
 - all takes ‘inclusive’ flag to include keys equals to *fromKey/toKey*
 - Supports reversed iteration
 - *descendingMap()* - returns a reversed view of the map
 - *descendingKeySet()* returns a reversed key *NavigableSet<K>*

Concurrent Collections

Concrete Concurrent Maps

- *ConcurrentSkipListMap<K,V>*
 - *ConcurrentNavigableMap* implementation
 - Uses SkipLists for fast search on sorted lists
 - Maintains hierarchical subsequences
 - Allows skipping range of irrelevant objects while searching

Challenge: maintain counters in a multithreaded environment

- Available solutions:
 - Dirty counters
 - Synchronization
 - RWLock via Lock API
 - Volatile
 - Atomic concurrent
 - Java 8 offers: LongAdders

- Available solutions:
 - Dirty counters
 - Reading and writing directly from & to static allocations
 - It's dirty since $x+=y$ is broken into several instructions by the JVM
 - **Useful for low integrity data reads**
 - Synchronization
 - It's cool, but basically – cancels parallelism.....
 - Threads might block for a long time in high concurrencies
 - **Useful for exclusive limited number of thread control**
 - RWLock via Lock API
 - Great for more sophisticated locks
 - Several monitors (lock flags)
 - Use multiple separated wait-notify mechanism
 - Manage concurrency according to thread contexts (Reader/Writer)
 - **Good mostly when there is a limited, known number of writer threads**

- Available solutions:
 - **Volatile**
 - Makes sure that any change made to a value is visible to all threads
 - Done on the instruction set level
 - In most cases cancels / prevents code optimization done by the JVM
 - **Useful when there's only one writer thread and multiple reader threads**
 - Multiple writer threads will cause race conditions just like with static...
 - **Atomic Concurrent**
 - Conditional update prevents race conditions
 - Means that thread in a race might re-try to acquire a lock and update
 - In high contention the thread might re-try for a very long time....
 - **Useful for small amounts of threads or limited amount of writer threads**
 - **Adders**

Adders

- In low concurrencies performs just like Atomic Concurrent
- But in high concurrencies performs MUCH better:
 - LongAdder holds a collection of cells. Each cell behaves like an AtomicLong
 - Threads in race conditions do not re-try. They place their update in a cell instead
 - Cells are evaluated every time we call for final result
- Good for frequent updates in high concurrency systems

```
//initialized with 0 by default
final LongAdder adder=new LongAdder();
Runnable task1 = ()->{ adder.add((long)(Math.random()*10000));};
Runnable task2 = ()->{ adder.increment();};
Runnable task3 = ()->{ adder.decrement();};
Runnable task4 = ()->{ adder.longValue();}
```



JAVA REFLECTION



Java Reflection



- Purpose
- Capabilities
- Examining classes and invoking object methods
- Manipulating objects
- Working with Arrays
- Java 1.5 enhancements

Purpose

- Represents, or reflects, the following in the running Java Virtual Machine:
 - Classes
 - Interfaces
 - Objects
- Useful for
 - manipulating known & unknown classes
 - writing development tools such as
 - Browsers
 - Debuggers
 - GUI builders
 - Code generators

Purpose

- Do not use for:
 - Creating method references to native methods
 - You should rather implement it in java
 - Problematic debugging and tracking
 - Wrap native code (prefer JNI)
- Note ! Reflection breaks encapsulation
 - Private fields are visible and accessible
 - Final fields can be populated

Capabilities

- Determine the class of an object
- Get information about a class's modifiers, fields, methods, constructors, and super-classes
- Describe primitives
- Find out what constants and method declarations belong to an interface
- Create an instance of a class whose name is not known until runtime
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime
- Invoke a method on an object, even if the method is not known until runtime
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components



Examining classes



- Why and when
- Retrieving Class objects
- Getting class name
- Checking if the Class reflects an Interface
- Getting class modifiers
- Identifying implemented Interfaces
- Examining the class fields
- Examining the class constructors
- Discovering class methods

Examining classes

- Required for class browsing
 - Need to know the class infrastructure
 - For example – list of class constructors
- Is done via *java.lang.Class*
 - The JRE maintains a *Class* object to reflect classes
 - *Class* instances are immutable
 - Also represent Interfaces
 - Each living object hold a reference to its *Class*

Examining classes

- Retrieving Class objects

- From an instance

```
MyObject obj=new MyObject();
Class class= obj.getClass();
```

- By specifying the class name as a *String*

- Checked at runtime
 - Portable

```
Class class= Class.forName("myPackage.MyObject");
```

- By specifying the class name at compile time

```
Class class= myPackage.MyObject.class;
```

Examining classes

- Getting class name

- Is done via the *getName()* & *toString()* methods
- If primitive is reflected – the name of the primitive is returned
- Returns a fully qualified name of the object/class reflected

```
import java.lang.reflect.*;
Import java.util.Date;

public class SampleName {
    public static void main(String[] args) {
        Date d = new Date();
        printName(d);
    }
    static void printName(Object o) {
        Class c = o.getClass();
        String s = c.getName();
        System.out.println(s);
    }
}
```

The sample program prints the following line: java.util.Date

Examining classes

- Checking if the Class reflects an Interface
 - Is done via *isInterface()* method

```
public static void verifyInterface(Class c) {  
    String name = c.getName();  
    if (c.isInterface()) {  
        System.out.println(name + " is an interface.");  
    } else {  
        System.out.println(name + " is a class.");  
    }  
}
```

Examining classes

- Identifying implemented Interfaces
 - Is done by the *getInterfaces()* method
 - Returns an array of Classes represents the Implemented interfaces

```
public static void printInterfaceNames(Object o) {  
    Class c = o.getClass();  
    Class[] theInterfaces = c.getInterfaces();  
    for (int i = 0; i < theInterfaces.length; i++) {  
        String interfaceName = theInterfaces[i].getName();  
        System.out.println(interfaceName);  
    }  
}
```

Calling this method with a RandomAccessFile instance will generate the following output:

java.io.DataOutput
java.io.DataInput

Note that the interface names printed are fully qualified

Examining classes

- Getting class modifiers
 - Is done via the *getModifiers()* method
 - The method returns an *int* value that can be examined using
java.lang.reflect.Modifiers class

```
public static void printModifiers(Object o) {  
    Class c = o.getClass();  
    int m = c.getModifiers();  
    if (Modifier.isPublic(m))  
        System.out.println("public");  
    if (Modifier.isAbstract(m))  
        System.out.println("abstract");  
    if (Modifier.isFinal(m))  
        System.out.println("final");  
}
```

The output of the sample method with a given *String* object reveals that the modifiers of the *String* class are public and final:

public
final

Examining classes

- Modifiers can be checked also on:
 - constructors
 - methods
 - data members
- Use method *isAccessible()* to check accessibility
- The method is available for any *AccessibleObject* implementer [Constructor, Method, Field]
- *AccessibleObject* also supports setting accessibility using the *setAccessible(boolean)* method
 - Enables or disables access check by the VM
 - May throw security exception
 - For final members – after calling *setAccessible()* on non-populated final field – it can be populated via reflection

Examining classes

Examining class fields

- Done via *getFields()* method that returns public Field array
- Done via *getDeclaredFields()* method that returns all Field array
- Field class provides the following abilities:
 - Getting field name – *getName()*
 - Getting field type – *getType()* which returns a *Class*
 - Set<type> method to assign a Field with a value to an Object
(is done on a field with the same name of this Field instance and type specified in the method name)
 - Get modifiers – *getModifiers()*
 - Get<type> to read value of static members from an object
(is done on a field with the same name of this Field instance and type specified in the method name)

Examining classes

Examining class fields - example

```
public static void printFieldNames(Object o) {  
    Class c = o.getClass();  
    Field[] publicFields = c.getFields();  
    for (int i = 0; i < publicFields.length; i++) {  
        String fieldName = publicFields[i].getName();  
        Class typeClass = publicFields[i].getType();  
        String fieldType = typeClass.getName();  
        System.out.println("Name: " + fieldName + ", Type: " + fieldType);  
    }  
}
```

The output of the sample program is:

Name: sum, Type: int

Name : word, Type: java.lang.String

Examining classes

Discovering class constructors

- is done for public via *getConstructors()* method
- is done for all via *getDeclaredConstructors()* method
- returns a Constructor array
- Constructor class supports:
 - getting constructor name
 - getting constructor parameters as *Class* array
 - creating new instance using the *newInstance(Object params)* method [where parameters are sent as objects and primitives are wrapped in wrapper classes]

Examining classes

Discovering class constructors - example

```
import java.lang.reflect.*;
import java.awt.*;
public class SampleConstructor {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        showConstructors(r);
    }
    public static void showConstructors(Object o) {
        Class c = o.getClass();
        Constructor[] theConstructors = c.getConstructors();
        for (int i = 0; i < theConstructors.length; i++) {
            System.out.print("(" );
            Class[] parameterTypes = theConstructors[i].getParameterTypes();
            for (int k = 0; k < parameterTypes.length; k++) {
                String parameterString = parameterTypes[k].getName();
                System.out.print(parameterString + " ");
            }
            System.out.println(")");
        }
    }
}
```

The output of the sample program is:
()
(int int)
(int int int int)
(java.awt.Dimension)
(java.awt.Point)
(java.awt.Point java.awt.Dimension)
(java.awt.Rectangle)

Examining classes

Discovering class methods

- is done for public via *getMethods()* method
- Is done for all via *getDeclaredMethods()* method
- returns a Method array
- Method class supports:
 - getting method name
 - getting method parameters as *Class* array
 - getting returned type
 - invoke – this method takes the instance to call and the *Object[]* of parameters to send. the method returns the returned value as an *Object*



Manipulating Objects



- Creating objects
- Getting fields values
- Setting fields values
- Invoking methods

Manipulating Objects

Creating objects

- Class supports a call to the default constructor

```
myPackage.MyObject obj=null;  
obj=(myPackage.MyObject)Class.forName("myPackage.MyObject").newInstance();
```

- When constructor expects arguments do the following:
 - load the required constructor as a Constructor
 - explore the constructor signature if needed/unknown
 - generate an object array with ordered parameters [where primitives values are wrapped in a matching wrapper class]
 - invoke the constructor using the Constructor's method:
newInstance(Object[] initArgs)

Manipulating Objects

Creating objects - example

```
public static void main(String[] args) {  
  
    Rectangle rectangle;  
    Class rectangleDefinition;  
    Class[] intArgsClass = new Class[] {int.class, int.class};  
    Integer height = 12;  
    Integer width = 34;  
    Object[] intArgs = new Object[] {height, width};  
    Constructor intArgsConstructor;  
  
    try {  
        rectangleDefinition = Class.forName("java.awt.Rectangle");  
        intArgsConstructor = rectangleDefinition.getConstructor(intArgsClass);  
        rectangle = (Rectangle) createObject(intArgsConstructor, intArgs);  
    } catch (ClassNotFoundException e) {  
        System.out.println(e);  
    } catch (NoSuchMethodException e) {  
        System.out.println(e);  
    }  
}  
....
```

Manipulating Objects

Example – cont.

```
public static Object createObject(Constructor constructor, Object[] arguments) {  
  
    System.out.println ("Constructor: " + constructor.toString());  
    Object object = null;  
  
    try {  
        object = constructor.newInstance(arguments);  
        System.out.println ("Object: " + object.toString());  
        return object;  
    } catch (InstantiationException e) {  
        System.out.println(e); //class is an interface or abstract so it cannot be instantiated  
    } catch (IllegalAccessException e) {  
        System.out.println(e); //constructor is private or protected  
    } catch (IllegalArgumentException e) {  
        System.out.println(e);  
    } catch (InvocationTargetException e) {  
        System.out.println(e); //this is a checked exception thrown by any 'invoke' operation  
    }  
    return object;  
}
```

The sample program prints a description of the constructor and the object that it creates:
Constructor: public java.awt.Rectangle(int,int)
Object: java.awt.Rectangle[x=0,y=0,width=12,height=34]

Manipulating Objects

Getting fields values

- Allows to dynamically read fields values
- Works also for static members
- Done with the following steps:
 - load a Class instance
 - receive the field representation using the *getField(String fieldName)* method
 - use matching get method (works also for static members)
 - for example:
if the Field instance represents a “size” data member
and the “size” is of type – float
than the method *getFloat(Object o)* should be called
getDouble(Object o) will also do the job
getBoolean(Object o) will throw an *IllegalArgumentException* in
this case

Manipulating Objects

Getting fields values - example

```
public class SampleGet {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(100, 325);  
        printHeight(r);  
    }  
    public static void printHeight(Rectangle r) {  
        Field heightField;  
        int heightValue;  
        Class c = r.getClass();  
        try {  
            heightField = c.getField("height");  
            heightValue = heightField.getInt(r);  
            System.out.println("Height: " + heightValue.toString());  
        } catch (NoSuchFieldException e) {  
            System.out.println(e);  
        } catch (IllegalAccessException e) {  
            System.out.println(e);  
        } catch (Exception e) {  
            System.out.println(e);  
        }  
    }  
}
```

The output of the sample program shows the value of the height field:
Height: 325

Manipulating Objects

Setting fields values

- Allows to dynamically update fields values
- Done with the following steps:
 - load a Class instance
 - receive the field representation using the *getField()* method
 - use matching set method (works also for static members)
 - for example:

if the Field instance represents a “size” data member
and the “size” is of type – *float*
than the method *setFloat(Object o, float value)* should be called
setBoolean(Object o, boolean value) will throw an *IllegalArgumentException*

Manipulating Objects

Setting fields values - example

```
public class SampleSet {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(100, 20);  
        System.out.println("original: " + r.toString());  
        modifyWidth(r, new Integer(300));  
        System.out.println("modified: " + r.toString());  
    }  
    public static void modifyWidth(Rectangle r, Integer widthParam ) {  
        Field widthField;  
        Class c = r.getClass();  
        try {  
            widthField = c.getField("width");  
            widthField.setInt(r, widthParam);  
        } catch (NoSuchFieldException e) {  
            System.out.println(e);  
        } catch (IllegalAccessException e) {  
            System.out.println(e);  
        }  
    }  
}
```

The output of the sample program shows that the width changed from 100 to 300:
original: java.awt.Rectangle[x=0,y=0,width=100,height=20]
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]

Manipulating Objects

Invoking methods

- Allows to dynamically invoke methods
- In order to dynamically invoke a method do the following:
 - load a *Class* instance
 - load the required method as a *Method*
 - explore the method signature if needed/unknown
 - generate an object array with ordered parameters
 - [where primitives values are wrapped in a matching wrapper class]
 - invoke the method using the *Method*'s method:
 - *invoke(Object o, Object[] Args)*

Manipulating Objects

Invoking methods - example

```
public static void main(String[] args) {
    String firstWord = "Hello ";
    String secondWord = "everybody.";
    String bothWords = append(firstWord, secondWord);
    System.out.println(bothWords);
}

public static String append(String firstWord, String secondWord) {
    String result = null;
    Class c = String.class;
    Class[] parameterTypes = new Class[] {String.class};
    Method concatMethod;
    Object[] arguments = new Object[] {secondWord};
    try {
        concatMethod = c.getMethod("concat", parameterTypes);
        result = (String) concatMethod.invoke(firstWord, arguments);
    } catch (NoSuchMethodException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    } catch (InvocationTargetException e) {
        System.out.println(e);
    }
    return result;
}
```

The output of the preceding program is:
Hello everybody.



Working with Arrays



- *java.lang.reflect.Array*
- Identifying arrays
- Retrieving components types
- Creating arrays
- Setting and getting arrays elements

java.lang.reflect.Array

provides static methods to do the following:

- dynamically instantiate an arrays
- get and set values according to array index
- get array length



Working with Arrays

- Identifying arrays
 - useful if we want to know if an object is an actual instance of an array
 - Use the *Class* method – *isArray()*
- Retrieving array type
 - Allows to dynamically reflect an array element type
 - done via *getComponentType()* method
 - returns a reflection (*Class*) of an array elements type or null if not an array

Working with Arrays

Getting array name and type:

```
public static void main(String [] args){  
    String [] strings={"hello","world"};  
    Object [] objects={"hello","world"};  
    int [] numbers ={1,2,3,4,5};  
    long [] longs ={1,2,3,4,5};  
    double [] doubles={1.1,2.2,3.3};  
    printArrayType(strings);  
    printArrayType(objects);  
    printArrayType(numbers);  
    printArrayType(longs);  
    printArrayType(doubles);  
}  
  
public static void printArrayType(Object target) {  
    Class targetClass = target.getClass();  
    if (targetClass.isArray()) {  
        String fieldName = targetClass.getName();  
        Class fieldType = targetClass.getComponentType();  
        System.out.println("Name: " + fieldName + ", Type: " + fieldType);  
    }  
}
```

Output:

```
Name: [Ljava.lang.String;, Type: class java.lang.String  
Name: [Ljava.lang.Object;, Type: class java.lang.Object  
Name: [I, Type: int  
Name: [J, Type: long  
Name: [D, Type: double
```

Working with Arrays

Creating arrays

- is done via `Array.newInstance()` method
- the method takes:
 - a component type as Class
 - length / size as an `int` or `int[]` for matrixes

```
public static Object doubleArray(Class componentType, int size) {  
    Object result = Array.newInstance(type,size);  
    return result;  
}
```

Working with Arrays

Setting and getting arrays elements

- in order to do so without reflection you'll need to specify the name of the array:

```
int [] number = new int[10];
numbers[0] = 100;
```

- Array class helps in setting and getting values using setters and getters methods that takes:
 - destination array
 - index
 - value

```
public static void copyArray(Object source, Object dest) {
    for (int i = 0; i < Array.getLength(source); i++) {
        Array.set(dest, i, Array.get(source, i));
        System.out.println(Array.get(dest, i));
    }
}
```

Java 1.5 enhancements

java.lang.Class

- Class supports generics which means it is type safe
- Class<T> - <T> stands for the represented class
- For example:
 - the type of *String.class* is *Class<String>*
 - the type of *Serializable.class* is *Class<Serializable>*
- The *newInstance()* method returns T

```
String s="Hello";
Class<? extends String> c=s.getClass();
String st=c.newInstance(); //no casting is needed
```

Java 1.5 enhancements

- Supports for Java Annotations
- *java.lang.reflect* has annotation support
- Every reflected entity has these 3 methods:
 - *public <T extends Annotation> T getAnnotation (Class< T extends Annotation > annotationClass)*
 - *public Annotation [] getDeclaredAnnotations()*
 - *public Annotation [] getAnnotations()*
 - Includes inherited annotations

Java 1.5 enhancements

An example:

```
public @interface Copyright {  
    String value();  
}
```

```
@Copyright (“John Bryce Training Center”)  
or  
@Copyright (value=“John Bryce Training Center”)  
public class Employee{  
....
```

```
..  
public static void main(String args) {  
    try{  
        Copyright copyright = Class.forName(“Employee”).getAnnotation(Copyright.class);  
        System.out.println(copyright.value());  
    }catch (ClassNotFoundException e){...}  
}..
```

Java 1.5 enhancements

- Supports for Java Enums
- *public boolean isEnum()*
 - Returns true only if the reflected entity is an Enum
- *public T[] getEnumConstants()*
 - Returns an array of the Enum values
 - Or null if the class doesn't represent Enum



JAVA NIO & NIO.2



- New I/O is an important addition to the java.io package
- Supported features:
 - Buffers for data of primitive types
 - Channels, a new primitive I/O abstraction
 - Character-set encoders and decoders
 - A file interface that supports locks and memory mapping
 - A multiplexed, non-blocking I/O facility for writing scalable servers

Buffers

Buffer main operations are:

- Static allocate – for creating a buffer with specified capacity
- array – converts the buffer into primitives array
- duplicate – generates a new buffer that holds a copy of the origin buffer data
- slice – generates a new buffer that holds a copy of the origin buffer data from the current position to the end
- get / put – assigns / returns single or multiple values to / from the buffer
- wrap – fills an array with values taken out of the buffer. the array is synchronized with the buffer content [called – ‘backed array’]
- mark / reset – allows re-iterating
- flip – ‘trims’ the buffer – current position becomes the buffers limit
- rewind – sets the positioning to zero
- remaining – return the number of remaining bytes

Buffer is an abstract class

- All of its extensions are also abstract:
 - *ByteBuffer*
 - supports *putInt/getInt*, *putFloat/getFloat*, etc..
 - Can be retrieved as *IntBuffer*, *FloatBuffer*, etc..
 - Can be retrieved as a read-only buffer
 - *CharBuffer*
 - *DoubleBuffer*
 - *FloatBuffer*
 - *IntBuffer*
 - *LongBuffer*
 - *ShortBuffer*
- Creating buffer is done by one of the following options:
 - Using the buffer static method *allocate (int capacity)*
 - Via channels

Channels

- A new primitive I/O abstraction
- represent an open connection to an entity like File or Socket
- has two states: open & close
- starts in open mode
- close operation is irreversible
- basic channel is *java.nio.Channel* interface with the following methods:
 - *close()*
 - *isOpen()*
- Channels are THREAD SAFE !

Channel sub interfaces are:

- *ReadableByteChannel* – add the *read(ByteBuffer)* method that reads data into the buffer
- *ScatteringByteChannel* – add *read(ByteBuffer [])* – useful for loading formatted data into different buffers
- *WriteableByteChannel* – add *write(ByteBuffer)* method that write the data from the buffer
- *GatheringByteChannel* – add *write(ByteBuffer [])* – useful for writing formatted data taken from different buffers
- *InterruptibleChannel* – overrides *close()* method so that:
 - When a thread is blocked in an I/O operation *close()* on the channel or *interrupt()* on the thread will cause the channel to close connection
 - The blocked thread will receive a *ClosedByInterruptException*

Concrete Channels

- *FileChannel* - A channel for reading, writing, mapping, and manipulating a file
- *SocketChannel* - A channel for stream socket connection
- *ServerSocketChanel* - A channel for stream socket connection and listening
- *SelectableChannel* – a channel that is managed by a Selector:
 - may determine whether I/O is blocking or non-blocking
[*configureBlocking(boolean)*]
 - Selector holds registered selectable channels
 - Selector can select the active channels and un-register those who are not
- *Pipe.SinkChannel* - A channel representing the writable end of a Pipe
- *Pipe.SourceChannel* - A channel representing the readable end of a Pipe
- Pipe - A pair of channels that implements a unidirectional pipe
 - Operations:
 - *open()*
 - *sink()* returns the *SinkChannel* of this pipe to perform write operations
 - *source()* returns *SinkSource* of this pipe to perform read operations

Concrete Channels creation is done via

- ‘Selector Provider’
 - Is a singleton
 - Has a no-argument constructor
 - Extends and implements the abstract class *SelectorProvider*
 - provides most methods for creating and opening channels:
 - *openDatagramChannel()*
 - *openPipe()*
 - *openSelector()*
 - *openServerSocketChannel()*
 - *openSocketChannel()*
- OLD I/O enhancements
 - FileChannel - returned by
 - *FileInputStream.getChannel()*
 - *FileOutputStream.getChannel()*
 - *RandomAccessFile.getChannel()*

Reading & writing form file example:

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class FileByteCopy {
    public static void main(String[] args) throws Exception {
        //loading first 1.6 Kb of a text file
        FileInputStream in = new FileInputStream("file.txt");
        FileChannel channel = in.getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(1600);
        channel.read(buffer);
        channel.close();
        in.close();
        //printing to screen
        byte[] data = buffer.array();
        for (byte value : data)
            System.out.print((char) value);
        //storing in other file
        FileOutputStream out = new FileOutputStream("file2.txt");
        channel = out.getChannel();
        buffer.rewind();
        channel.write(buffer);
        channel.close();
        out.close();
    }
}
```

EXERCISE



Lab 6 – New I/O

In this exercise you will use I/O between two threads

Character-set encoders and decoders

Charset provides the following :

- list of names and aliases (like nickname) of the supported char-sets
- all supported charset in a Map collection

[SortedMap <String charsetName><Charset object>]

- creating encoders – in order to convert string to bytes
- creating decoders – in order to build strings from bytes

Character-set encoders and decoders

Supported charsets

- US-ASCII Seven-bit ASCII - the Basic Latin block of the Unicode character set
- ISO-8859-1 ISO Latin Alphabet No. 1 - ISO-LATIN-1
- UTF-8 Eight-bit Unicode Char Set Transformation Format
- UTF-16BE Sixteen-bit Unicode Char Set Transformation Format
- UTF-16LE Sixteen-bit Unicode Char Set Transformation Format
- UTF-16 Sixteen-bit Unicode Char Set Transformation Format
- OS supported charsets

Character-set encoders and decoders

CharsetEncoder

- An engine that can transform a sequence of sixteen-bit Unicode characters into a sequence of bytes in a specific charset
- constructor takes:
 - the Charset that creates & uses this encoder
 - average numbers of bytes that are generated per char
 - maximum numbers of bytes that may be generated per char
- main operation: encode
 - takes *CharBuffer*
 - returns *ByteBuffer*

Character-set encoders and decoders

CharsetDecoder

- An engine that can transform a sequence of bytes in a specific charset into a sequence of sixteen-bit Unicode characters
- constructor takes:
 - the Charset that creates & uses this decoder
 - average numbers of chars that are included per byte (float)
 - maximum numbers of chars that may be included in a single byte
- main operation: decode
 - takes *ByteBuffer*
 - returns *CharBuffer*

Character-set encoders and decoders

- Getting Charset instance can be done in several ways:
 - *Charset.defaultCharset()* – determined according to locale & OS properties
 - *Charset.forName(String charset)* – returns the Charset of the specified charset
 - *CharsetProvider.charsetForName(String charsetName)* – same as previous
- Checking if charset is supported:
 - *Charset.isSupported(String charsetName)*
 - *Charset.availableCharsets()* (returns SortedMap<String, Charset>)
 - *CharsetProvider.charsets()* (returns Iterator)

```
import java.nio.charset.Charset;  
  
public class CharsetExample {  
  
    public static void main(String[] args) throws Exception {  
        System.out.println(Charset.defaultCharset());  
        System.out.println(Charset.availableCharsets());  
        System.out.println(Charset.isSupported("ISO-8859-8"));  
    }  
}
```

Output:

```
windows-1252  
{Big5=Big5, Big5-HKSCS=Big5-HKSCS, ...  
true
```

Character-set encoders and decoders

- Java I/O support for NIO
- *FileInputStream*, *FileOutputStream* & *RandomAccessFile* provides
 - *getChannel()* method that returns *FileChannel*
 - updated *close()* operation that also closes the channel
- *InputStreamReader* & *OutputStreamWriter* provides
 - a constructors that takes a Charset object
 - *InputStreamReader* can take also a *CharsetDecoder*
 - *OutputStreamWriter* can take also a *CharsetEncoder*
- Same goes to *FilterInputStream* & *FilterOutputStream* decorator base classes

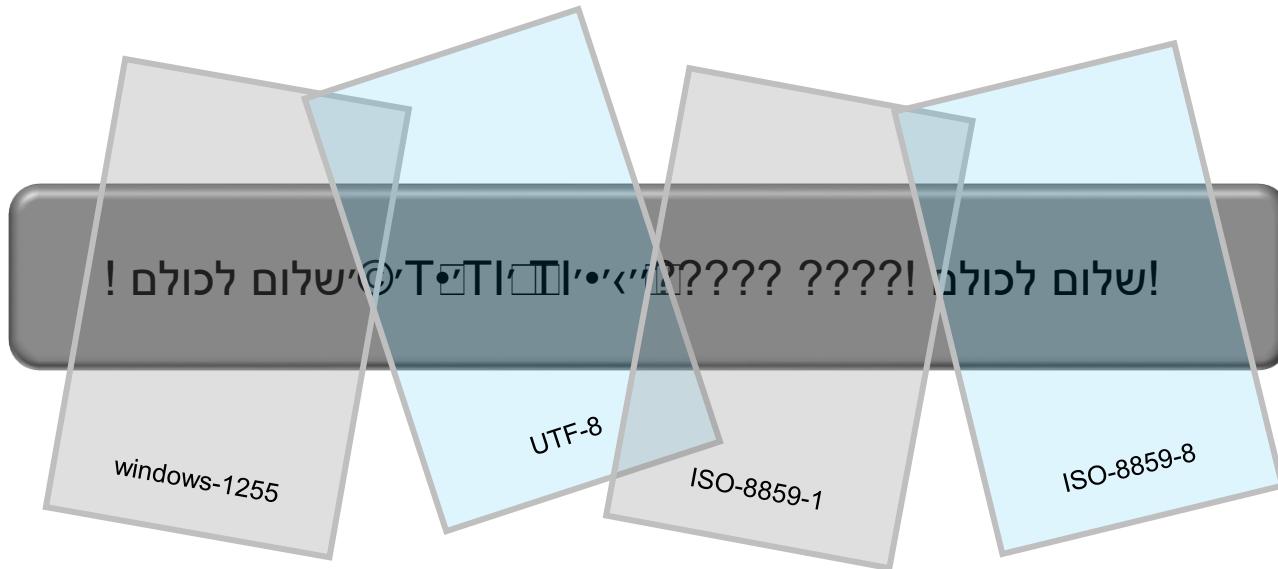
Character-set encoders and decoders

Example:

```
public class FileCharsetExample {  
  
    public static void main(String[] args) throws Exception {  
  
        Charset win1255=Charset.forName("windows-1255");  
        Charset utf8=Charset.forName("UTF-8");  
        Charset iso1=Charset.forName("ISO-8859-1");  
        Charset iso8=Charset.forName("ISO-8859-8");  
  
        String msg="שלום לכולם!\n";  
  
        storeEncodedBytes(win1255.encode(msg));  
        storeEncodedBytes(utf8.encode(msg));  
        storeEncodedBytes(iso1.encode(msg));  
        storeEncodedBytes(iso8.encode(msg));  
    }  
  
    private static void storeEncodedBytes(ByteBuffer encodedMsg) throws Exception{  
        RandomAccessFile raf=new RandomAccessFile("encoded.txt","rw");  
        FileChannel channel=raf.getChannel();  
        //appending data  
        raf.seek(raf.length());  
        channel.write(encodedMsg);  
        channel.close();  
        raf.close();  
    }  
}
```

Character-set encoders and decoders

Output file:



Added in Java 7

- Includes:
 - New File System API
 - File change automation
 - Asynchronous IO

File System API

- NIO Enhancements for File System API – NIO.2
 - `java.nio.file` (& sub-packages: `attribute` & `spi`)
 - Provides an extensible file system implementations
 - `SystemProvider` extensions can focus on JAR/ZIP and others
- What is wrong with existing Java FS solution ?
 - All in one class – `java.io.File`
 - Usage is not clear
 - is it abstract ? real ? both ?
 - Is it a file ? directory ? absolute directory ? partition ?
 - Many modern FS features are missing

Main services:

- FileSystems – FS interface. Manages access to files on the local file system
 - delete(path), copyFile(..),moveFile(..), getFileStores()...
 - isOpen(),isReadOnly(), isSameFile(path1, path2)...
- FileRef – provides methods to read/write to a file. Can also change file attributes
 - Obtaining: File.toPath(), URL.toFileRef()
 - get/setAttribute(), newInputStream(), newOutputStream()...
 - Using:

```
file.setAttribute("dos:hidden", true);
```

File System API

Main Services cont. – NIO.2

- Paths –points to a file/directory with system dependant path
 - compareTo(path), equals(path)
 - getNameCount(), iterator(), normalize() (get rid of '.' & '..')
 - toFile(), toAbsolutePath(), relativize() (creates relative path between this & the given)
- FileStore – reflects file storage area (like pool, device, partition, volume, OS file system)
 - getTotalSpace(), getUnallocatedSpace(), getUnusableSpace()
- Files - utility class
 - create(), copy(), delete()
 - exists(), getLastModified(), getAttribute(), isHidden(), isExecutable(), isDir()
 - getFileStore() , getOwner () (returns UserPrincipal)

File System API

- NIO Enhancements for File System API – NIO.2
 - Simple example:

```
Path src=Paths.get("C:/temp/data.bkp");
Path target=Paths.get("E:/info/data.bkp");

Files.copyTo(src,target, StandardCopyOption.REPLACE_EXISTING);
```

File Change Auto

- File change automation
 - Allows listening to events on directories
 - Events are detailed in StandardWatchEventType ENUM
 - ENTRY_CREATE, ENTRY_DELETE, ENTRY MODIFY
 - Use WatchService to obtain service
 - Use WatchKey to register service to a path (directory)
 - Use WatchEvent to obtain event count, type and context
 - Flow:
 - When event is detected the key is signaled
 - Signaled key enters to a watch service queue
 - Until consumed, the key might experience more events
 - All key events get handled when key is consumed from the queue

File Change Auto

- File change automation – NIO.2
 - Example:

```
WatchService ws = FileSystems.getDefault().newWatchService();
// we'll add StandardWatchEventKinds ENUM of ENTRY_DELETE
WatchKey key = someDirPath.register(ws, ENTRY_DELETE);
while(true){
    key=ws.take(); //blocks until key is signalled
    List<WatchEvent<?>> le=key.pollEvents();
    // process all deletion events
    key.reset();    //out of the queue, back to initial state
}
```

Asynchronous IO

- Goal: connect, read and write will be forked transparently
- Supported for files and sockets
 - AsynchronousFileChannel
 - AsynchronousSocketChannel, AsynchronousServerSocketChannel
- Can be implemented in 2 ways:
 - Initiate I/O operation that results with Future [use Executor-Callable]
 - Register CompletionHandler when invoking I/O operation [use Observer]

Asynchronous IO

Example using Future:

- Results in Future for connect, read & write operations

```
AsynchronousSocketChannel ch = AsynchronousSocketChannel.open();
Future<Void> result = ch.connect(new InetSocketAddress(ip,port));
....
// Future.get() return null on success
if(result.get()==null){
    ByteBuffer b = ....
    // wait read operation to complete
    Future<Integer> data = ch.read(b);
    ...
}
```

Asynchronous IO

Example using CompletionHandler:

- Handles connect, read & write operations

```
ByteBuffer b = ....  
// read using the buffer.  
// we may assign an attachment - attachments are returned on  
// completion. We don't need one here so we assign null  
ch.read(b,null,new CompletionHandler<Integer(Void>){  
    public void completed(Integer result, Void attachment ){  
        //process result data  
    }  
    public void failed(Throwable ex, Void attachment){  
        //process failure  
    }  
});
```



FUNCTIONAL PROGRAMMING WITH STREAMS API

Functional Interfaces

Lambdas and Functional Interfaces

- Java 7 Dynamic Invocation
- JSR 292: Java Platform Support for Dynamically Typed Languages
 - Engines can use additional runtime instructions in order to compile code
 - Invokedynamic – new bytecode instruction that allows to translate method invocation without relating to the object holds it
 - Java.dyn.MethodHandler – bounded to each Invokedynamic instruction
 - holds reference to a JVM method
 - cached
 - Both invokedynamic and MethodHandlers are updated during runtime
 - Are addition to specialInvoker (direct call), staticInvoker, virtualInvoker (abstract call) and interfaceInvoker
- That's great...we got this dynamic thing in our plugin but we can't use it in Java...

Functional Interfaces

Lambdas and Functional Interfaces

- Java 8 came to the rescue
 - Provides support for LAMBDA expressions
 - All LAMBDA calls are done via native invokedynamic
 - Now we can assign some code as a method
 - No need in callbacks (event model)
 - Reduces anonymous classes
 - More up-to-date way of coding

Functional Interfaces

Lambdas and Functional Interfaces

- LAMBDA expressions are supported for **functional interfaces**
- Functional interface
 - Denoted with **@FunctionalInterface**
 - Some interfaces were updated (Runnable, Comparable, Event....)
 - Many new interfaces are provided (later)
 - You may create your own

Functional Interfaces

Lambdas and Functional Interfaces

- Working with Functional Interfaces
- Old code
 - Abuse classes
 - Uses anonymous classes

```
public class MyRunnable implements Runnable {  
    public void run(){  
        .....  
    }  
}  
  
(new Thread(new MyRunnable())).start();
```

```
(new Thread(new Runnable(){  
    public void run(){  
        .....  
    }  
}).start();
```

Functional Interfaces

Lambdas and Functional Interfaces

- Since Runnable is now:
- You may use it like that:

```
@FunctionalInterface  
public interface Runnable {  
    public void run();  
}
```

```
Runnable r = () -> { .....  
};  
(new Thread(r)).start();
```

```
(new Thread(  
    () -> { ..... }  
)).start();
```

- If your method is single lined you may skip { }
- Not just simple coding... don't forget it is also executed dynamically !

Functional Interfaces

Lambdas and Functional Interfaces

- Single-lined & blocks examples:

```
Runnable r = () -> System.out.println("Hey!");  
(new Thread(r)).start();
```

```
Runnable r = () -> {  
    for(int i=0;i<100;i++){  
        System.out.println( i );  
    };  
(new Thread(r)).start();
```

```
(new Thread(() -> System.out.println("Hey!"))).start();
```

```
(new Thread(() -> {  
    for(int i=0;i<100;i++){  
        System.out.println( i );  
    }  
}).start();
```

Functional Interfaces

Behind the scene steps for execution:

Functional:

1. Method is allocated & interpreted
2. Thread object executes the method directly

The old way:

1. MyRunnable.class is loaded
2. MyRunnable instance is allocated on heap – this one is heaviest
3. Thread object executes the run() method by using the object reference

Both method & object.run() gets optimized

And cached when becomes hot

```
Runnable r = () -> System.out.println("Hey!");  
(new Thread(r)).start();
```

```
public class MyRunnable implements Runnable {  
    public void run(){  
        .....  
    }  
}
```

```
(new Thread(new MyRunnable())).start();
```

Functional Interfaces

What about ‘this’ keyword in LAMBDA exp?

- LAMBDA code is not executed from an object instance
 - Only the method gets loaded – so there is nothing but the method reflection (e.g Runnable.class → run())
 - This means it is treated & executed as a ‘static’ like content
 - So.....no ‘this’.....
-
- The following code will fail to compile with “cannot use this in a static content” error

```
Runnable r = () -> System.out.println(this);  
(new Thread(r)).start();
```

Functional Interfaces

Lambdas and Functional Interfaces

- What about methods that accept parameters ?
 - Type safety is determined & enforced during compile time
 - Example:

```
List<String> words =
    Arrays.asList("David", "Adam", "Eve", "Moses");
    Collections.sort(words, XXX);
```

 - here, the compiler assumes you're about to assign Comparator<String>
 - Comparator is now @FunctionalInterface
 - Therefore, the compiler checks any assigned function to have the following signature:

```
public int XXX (String s1, String s2)
```
- Just like Comparator<T>.compare(T,T) signature

Functional Interfaces

Lambdas and Functional Interfaces

```
public int XXX (String s1, String s2)
```

- So, we may come up with our own implementations
- But this time Instead of creating a class & force object instantiation...
- Old coding:

```
public class ReverseNameComparator implements  
Comparator<String>{  
  
    public int compare (String s1, String s2){  
        return s1.compareTo(s2)*(-1);  
    }  
}
```

Functional Interfaces

Lambdas and Functional Interfaces

```
public int XXX (String s1, String s2)
```

- We can do it this way:

```
List<String> words = Arrays.asList("David", "Adam", "Eve", "Moses");
Comparator<String> c=(String s1, String s2)->return s1.compareTo(s2)*(-1);
Collections.sort(words, c);
```

- Or this way:

```
List<String> words = Arrays.asList("David", "Adam", "Eve", "Moses");
Collections.sort(words, (String s1, String s2)->return s1.compareTo(s2) *(-1));
```

- Or this way – counting on Generics to evaluate s1, s2 types:

```
List<String> words = Arrays.asList("David", "Adam", "Eve", "Moses");
Collections.sort(words, (s1,s2)->return s1.compareTo(s2) *(-1));
```

Functional Interfaces

Lambdas and Functional Interfaces

```
public int XXX (String s1, String s2)
```

- Polymorphism isn't checked on dynamic invocation (only method signature)...
- This means we don't even need to implement Comparator (!)

```
public class Things {  
  
    public int doThings(String s1, String s2){  
        return s1.compareTo(s2)*(-1);  
    }  
}
```

- But how exactly can we assign it to Collections.sort() method ?? In a moment...

Functional Interfaces

Lambdas and Functional Interfaces

- You may create custom abstractions based on functional interfaces

```
@FunctionalInterface
public interface IncrementByOne<T> {
    public int increment();
}
```

```
public class RaiseValues<T> {
    public void raise(IncrementByOne<? super T> entity){
        System.out.println(entity.increment());
    }
}
```

- As we'll see later, any method with this signature :
can be assigned to RaiseValue.raise()

```
public int XXX ()
```

Functional Interfaces

Lambdas and Functional Interfaces

- Important functional interfaces
 - Existing & enhanced interfaces
 - *Runnable.run() : void*
 - *Comparator<T>.compare(T, T) : int*
 - Java provides several new Functional Interfaces for other purposes:
 - *Predicate<T>.test(T) : boolean*
 - *Consumer<T>.accept(T) : void*
 - *Supplier<T>.get() : T*
 - *Function<T,R>.apply(T) : R*

Functional Interfaces

Lambdas and Functional Interfaces

- Important functional interfaces
 - *Predicate<T>.test(T) : boolean*
 - Accepts T and calculate a boolean result. True = passed the test
 - *Consumer<T>.accept(T) : void*
 - Accept T and perform an operation. No result
 - *Supplier<T>.get() : T*
 - Produces T. Therefore accepts no parameters and returns T
 - *Function<T,R>.apply(T) : R*
 - Maps T value to R. Accepts T and calculate result R

Functional Interfaces

Lambdas and Functional Interfaces

- Functional interfaces cont.
 - More concrete interfaces for working with primitives and references
 - *IntPredicate.test(int) : boolean* Tests int value and result with boolean
 - *IntConsumer.accept(int) : void* Accepts int
 - *IntSupplier.getAsInt() : int* Produces int
 - *IntFunction<R>.apply(int) : R* Accepts int and produces Result R
 - Same exists for Double
 - *DoublePredicate, DoubleConsumer, DoubleSupplier, DoubleFunction*
 - And for Long
 - *LongPredicate, LongConsumer, LongSupplier, LongFunction*

Functional Interfaces

Lambdas and Functional Interfaces

- Functional interfaces cont.
 - Bi – functional interfaces that reduces two inputs, T and U, into one result
 - *BiPredicate<T,U>.test(T,U) : boolean*
 - *BiConsumer<T,U>.accept(T, U) : void*
 - *BiFunction<T,U,R>.apply(T,U) : R*

Method reference

- If we implement functional interface we use () -> { }
- But if we got a ready to use implementation that matches with its signature ?
 - We would like to reference and assign it
 - We would like to reuse it
 - Whether static or not, we would like to reference it through its class
- Well, this is how you reference methods:
 - Non-static:

```
SomeClass c=new SomeClass();
c :: someMethod
```

- Static :

```
SomeClass :: someMethod
```

Method reference

- Back to our examples: Collection.sort(List<String>,)

```
public int XXX (String s1, String s2)
```

- And we have this code, which doesn't implements Comparator but got a method with a matching signature to compare() named *doThings()*:

```
public class Things {  
  
    public int doThings(String s1, String s2){  
        return s1.compareTo(s2)*(-1);  
    }  
}
```

- Lets use method referencing to compare with *doThings()*:

```
List<String> words = Arrays.asList("David","Adam","Eve","Moses");  
Things t=new Things();  
Collections.sort(words, t::doThings);
```

Method reference

- Now, with a static method: `Collection.sort(List<String>,)`

```
public int XXX (String s1, String s2)
```

- In this case `doThings()` is static:

```
public class Things {  
  
    public static int doThings(String s1, String s2){  
        return s1.compareTo(s2)*(-1);  
    }  
}
```

- We use method referencing without instantiating an object:

```
List<String> words = Arrays.asList("David", "Adam", "Eve", "Moses");  
Collections.sort(words, Things::doThings);
```

Method reference

- Back to our custom functional interface example:

```
@FunctionalInterface  
public interface IncrementByOne<T> {  
    public int increment();  
}
```

```
public class RaiseValues<T> {  
    public void raise(IncrementByOne<? super T>  
entity){  
        System.out.println(entity.increment());  
    }  
}
```

- As mentioned, any method with this signature :
can be assigned to RaiseValue.raise()

```
public int XXX ()
```

Method reference

- So, here is a Person implementation that contains such a method:

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() {return name;}  
    public void setName(String name) {this.name = name;}  
    public int getAge() {return age;}  
    public void setAge(int age) {this.age = age;}  
  
    //NOTE: method signature matches IncrementByOne.increment() method  
    public int birthday(){  
        this.age+=1;  
        return this.age;  
    }  
}
```

Method reference

Usage:

```
Person p = new Person("David",20);
RaiseValues<Person> calc=new RaiseValues<Person>();
calc.raise(p::birthday);
```

Output : (since raise() prints increased value)

21

Method reference

- What if we would like to assign constructors ?
 - Java provides a functional interface that may be good choice for default constructors
 - *java.util.function.Supplier<T>* with the method: *T get()*
 - Example:

```
public class Factory<T> {  
    public T getObject(Supplier<T> s){  
        return s.get();  
    }  
}
```

```
Factory<Person> factory=new Factory<Person>();  
factory.getObject(Person::new);
```

Method reference

- Method reference performs better than LAMBDAs

```
test(p->p.getAge>18)..
```

```
test(Person::isMature)..
```

- But what if we need more complex and extensible calculations on our Person class?
 - Java Beans should be just value objects....
 - Solution: use Utility Classes

```
test(PersonUtils::isMature)..
```

```
public class PersonUtils{  
  
    public static boolean isMature (Person p)  
    {..}  
    ...  
}
```

Interface Default and Static Methods

- Interface can have default implemented methods
 - Unlike abstract methods – default doesn't require overriding
 - Can be overridden in implementation classes

```
public interface Dimensional {  
    int getWidth();  
    int getHeight();  
    default int getArea(int width, int height){  
        return width*height;  
    }  
    boolean resize() ;
```

Interface Default and Static Methods

- Interface can also have implemented static methods

```
public interface PersonFactory{  
    static Person getPerson( Supplier<Person>  
s){  
        return s.get();  
    }  
}
```

```
PersonFactory.getPerson(Person::new);
```

Interface Default and Static Methods

- What about ‘multiple inheritance’ potential collisions ?
 - Given these 3 interfaces:

```
public interface Dimensional {  
    ...  
    default int getArea(int w, int h){  
        return w*h;  
    } ...
```

```
public interface TriangleDimensional {  
    ...  
    default int getArea(int w, int h){  
        return (w*h)/2;  
    } ...
```

```
public interface AbstractDimensional {  
    ...  
    int getArea(int w, int h); //abstract  
    ...
```

Interface Default and Static Methods

```
public class MyShape1 implements Dimensional, TriangleDimensional{  
    ???  
}
```

- Which method is taken ?
 - Answer is that NONE is taken. When you get default collision it fails to compile.
- And in this case, does default methods override abstract methods ?

```
public class MyShape2 implements Dimensional, AbstractDimensional{  
    ???  
}
```

- NO. abstract methods NEVER gets overridden by default, so it fails to compile
- But – you may override the collided method

Interface Default and Static Methods

- Means that this code will compile:

```
public class Rectangle implements Dimensional, AbstractDimensional {  
  
    @Override  
    public int getArea(int w, int h) {  
        return w*h;  
    }  
    ....
```

- When default and abstract methods collide – you must provide your own implementation
- Changes to an existing interfaces might break code

Interface Default and Static Methods

- Keep in mind that
 - Changes to an existing interfaces might break code
 - At least there is no scenario in which you are running different code than what you think..
 - This is because
 - Multiple default method collision will not compile
 - Default & abstract method collision will force you to override

Interface Default and Static Methods

- What about multiple method Functional Interfaces ?
- Functional Interfaces are meant to assign a method
- So, only one method can be abstract
- Others must be default or static
- This is useful in cases where several methods are invoked in some order along with the assigned one

```
@FunctionalInterface
public interface CalculationFlow{
    default void logTx(){...}
    default void recordTx(){....}
    void calculate(Consumer<T> c)
}
```

Interface Default and Static Methods

- Possible multiple method invokers:
- When providing CalculatorFlow implementations both record() & log() may be overridden

```
public void nonRecordedCalc(CalculationFlow cf){  
    cf.calculate((v)->{.....});  
}  
  
public void recordedCalc(CalculationFlow cf){  
    ct.recordTX();  
    cf.calculate((v)->{.....});  
    cf.logTX();  
}
```

Handling null Results

Optional

- Main goal is to clear our code from null checks
- java.util.Optional
- An object container that provides comfortable behavior regarding nulls & voids
 - Creating Optional

```
Optional<String> eValue=Optional.empty();  
Optional<String> nValue=Optional.ofNullable(null);  
Optional<String> value=Optional.of("OptionalData");
```

- Empty Optional is a void container
- Nullable Optional may contain value (T) or nulls
- Other holds values (T)

Handling null Results

Optional

- Basic behavior
 - get() – results in T or NoSuchElementException for empty & nullable Optional
 - orElse(T) – results in wrapped T if present, else return assigned T
 - orElseGet(Supplier<? extends T>) – results in T or executes Supplier<T>

```
System.out.println(value.get());  
System.out.println(nValue.orElse("else None"));  
System.out.println(eValue.orElse("else Empty"));  
//Supplier<String>  
System.out.println(value.orElseGet(()->"else-get ???"));  
System.out.println(nValue.orElseGet(()->"else-get  
None"));  
System.out.println(eValue.orElseGet(()->"else-get  
Empty"));
```

Supplier<T> functional interface
get() : T

Output
OptionalData
else None
else Empty

OptionalData
else-get None
else-get Empty

Handling null Results

Optional

- Basic behavior
 - `isPresent()` – results true if Optional wraps an object or false otherwise
 - `ifPresent(Consumer<? super T>)` – if object is present, executes `Consumer<T>.accept(T)` assigning wrapped object T and result with void

```
System.out.println(value.isPresent());
System.out.println(nValue.isPresent());
System.out.println(eValue.isPresent());

//Consumer
value.ifPresent((String s)->System.out.println(s+" is in the
house));
```

Consumer<T> functional interface
`accept(T) : void`

Output

true
false
false

OptionalData is in the
house

Handling null Results

Optional

- Primitives Optionals
 - OptionalDouble, OptionallInt, OptionalLong
 - Let have a look at OptionalDouble, others are just the same idea..
 - Everything turns to be double oriented:
 - of(double) – returns OptionalDouble wrapping the given value
 - get() becomes getAsDouble() and results with the wrapped double value or null
 - orElse(double) – return wrapped double if present. If not – returns the assigned double
 - orElseGet (DoubleSupplier) – same but if not present invokes getAsDouble() : double
 - ifPresent(DoubleConsumer) – if present, assigns value to accept(double) and returns void

Java Streams API

- New `java.util.stream` API
- A sequence of elements supporting sequential and parallel aggregate operations
- Mostly Relevant for dealing with huge data grids
- Therefore obtained mainly from Collections, but not only
- Introduces real-world functional-style programming
- Since it is functional based - it is much faster

Java Streams API

- Streams perform lazily. Means the code is evaluated only if must be executed
- All manipulations are put in a stream pipeline that can be consumed, collected or manipulated again
- If multiple threads are used – each has its own pipeline which the stream gathers
- Spliterators are used behind the scene for breaking huge processing into small parts
- When processing completes – we are at the end of the stream. Cannot re-iterated

Streams vs. Collections

- Collections uses External iteration
 - Means you should handle elements in-between iterations
 - All operations are eagerly executed
- Streams uses Internal iterations
 - Means that elements are handled by hidden holders in between iterations if & when needed
 - Operations are executed lazily – when terminal operation is triggered (later)

Java Streams API

Best in this case is to cover features through code

- For the next examples we'll use the following:
 - Address class – Java Bean with City enum & street attributes
 - Person class – Java Bean with name, age, Gender enum & Address
 - List<Person> populated with several thousands Person instances
 - We'll do some cool manipulations on List<Person> with stream API

Java Streams API

Example POJOs:

```
public class Person {  
    private String name;  
    private int age;  
    private Address address;  
    private Gender gender;  
    public Person(String name, int age, Gender gender, City city, String street) {  
        this.name = name;  
        this.age = age;  
        this.gender=gender;  
        address=new Address(city, street);  
    }  
    public String getName() {return name;}  
    public void setName(String name) {this.name = name;}  
    public int getAge() {return age;}  
    public void setAge(int age) {this.age = age;}  
    public Address getAddress() {return address;}  
    public void setAddress(Address address) {this.address = address;}  
    public Gender getGender() {return gender;}  
    public void setGender(Gender gender) {this.gender = gender;}  
    @Override  
    public String toString() {return name + "-" + age;}  
}
```

Java Streams API

Example POJOs & Enums:

```
public enum City {  
    A,B,C,D,E,F,G,H  
}
```

```
public enum Gender {  
    M,F  
}
```

```
public class Address {  
    private City city;  
    private String street;  
  
    public Address(City city, String street) {  
        this.city = city;  
        this.street = street;  
    }  
    public City getCity() {return city;}  
    public void setCity(City city) {this.city = city;}  
    public String getStreet() {return street;}  
    public void setStreet(String street) {this.street = street;}  
    @Override  
    public String toString() {return city.toString();}  
}
```

Java Streams API

Loading lots of persons to memory

```
List<Person> people = new ArrayList<>();
for (int i = 0; i < 10000; i++) {
    int cityInx = (int) (Math.random() * 8);
    int genderInx = (int) (Math.random() * 2);
    int letterInx = (int) (Math.random() * 24);
    people.add(new Person("!" + (char) ('a' + letterInx) + "!",
        (int) (Math.random() * 121), Gender.values()[genderInx],
        City.values()[cityInx], (char) i + ".St"));
}
```

Java Streams API

- Obtaining streams and parallel streams

```
List<Person> people = new ArrayList<>();  
....  
Stream<Person> stream=people.stream();  
Stream<Person> parallel=people.parallelStream();  
Stream<Person>.of(new Person(..),new Person(..),...);
```

- Arrays utility class provides array based streams

```
int [] nums =new int[100000];  
....  
IntStream= Arrays.stream(nums)
```

Basic collective operations with streams

- *count()* – counts elements in the stream
- *distinct()* – returns stream of unique values (uses Object.equals())
- *empty()* – returns an empty stream
- *findAny()* – peeks (randomly) a value from the stream and returns it wrapped in Optional
- *findFirst()* – does the same but always with the first element in the stream
- *limit(long maxSize)* – returns a sub-stream from first element to maxSize
- *skip(long n)* – returns a sub-stream starting from n to last element
- *sorted()* – returns a stream of sorted element – sorts according to naturally ordering
- *sorted(Comparator<? super T> comparator)* – does the same but sorts with compare(T t1,T t2)
- *iterator()*

Basic collective operations with streams

- Simple example with array stream:

```
int [] nums =new int[100000];
for(int i=0;i<nums.length;i++){
    nums[i]=(int)(Math.random()*100000);
}

System.out.println(Arrays.toString(nums));
int[] sorted=Arrays.stream(nums).sorted().toArray();
System.out.println(Arrays.toString(sorted));
System.out.println( Arrays.stream(nums).average().getAsDouble());
```

Basic collective operations with IntStreams, LongStreams & DoubleStreams

- Can save the headache when handling index loops with ints, longs and doubles
 - range(long s, long e) – returns a sub-stream from s to e (e is excluded)
 - rangeClosed(long s, long e) – returns a sub-stream from s to e (e is included)
 - iterate() – dynamically streams value based on previous value
 - Accepts initial value and applyAsInt(int):int dynamic method
 - Comes from IntUnaryOperator functional interface
 - Also available for long & double
 - Iterates infinitely - use limit()
 - generate() – dynamically generates a stream with a given Supplier
 - Generates infinitely – use limit()
 - limit(long maxSize) – provides a stream with the given maxSize

Java Stream API's

Basic collective operations with streams

- Simple example with int stream:

```
IntStream.range(1,4);  
IntStream.rangeClosed(1,4);
```

Stream contains: 1,2,3

```
IntStream ist= IntStream.of(1,2,3,4,5,6);  
ist.limit(3);
```

Stream contains: 1,2,3,4

```
IntStream.iterate(0, (i) -> i+2).limit(10); //start with zero, increment index in 2  
IntStream.generate(()->(int)(Math.random()*10).limit(100));
```

Stream contains: 1,2,3

Stream contains:
0,2,4,6,8,10,12,14,16,18

Match operations with streams

- allMatch(Predicate <? super T>) - returns true if all elements in the stream passes the test
- anyMatch(Predicate <? super T>) – return true if one element in the stream passes the test
- noneMatch(Predicate <? super T>) – return true if no element in the stream passes the test

Predicate <T> functional interface
public boolean test (T)

```
List<Person> people = new ArrayList<>();
//fill with 10,000 person instances
....
System.out.println(people.stream().allMatch(p->p.getName().startsWith("!")));
System.out.println(people.stream().allMatch(p->p.getName().startsWith("!h")));
System.out.println(people.stream().anyMatch(p->p.getName().startsWith("!h")));
System.out.println(people.stream().anyMatch(p->p.getName().startsWith("!hx")));
```

Output
true
false
true
false

Java Streams API

Match operations with streams

- Using predefined tests :

```
List<Person> people = new ArrayList<>();  
//fill with 10,000 person instances
```

....

```
System.out.println(people.stream().anyMatch(MaturePredicate::isMature));  
System.out.println(people.stream().allMatch(MaturePredicate::isMature));
```

```
public class MaturePredicate{  
    //could be a Person method as well...  
    public static boolean isMature(Person t) {  
        if(t.getAge()>=18) return true;  
        return false;  
    }  
}
```

Output

true
false

Java Streams API

Lazily executed stream operations:

- *filter(...)*
- *map(...)*
- *peek(...)*
- *flatMap(...)*
- *empty()*
- *distinct()*
- *limit(...)*
- *skip(...)*
- *sorted(), sorted(...)*

Streams - Lazily executed stream operations

- Think of that every time an intermediate operation is added to a stream it is actually being added to the stream pipeline
- The stream pipeline is executed when terminal operation is invoked
- Each element in the stream source is passed through the pipeline
 - Result contains only those who passed all the way
 - Element that fails is not evaluated along the rest of the pipeline
 - Single iteration over the stream source
- Pipelines iterate over stream sources and therefore cannot be reused

Streams - Lazily executed stream operations

- Interfering
 - Basically, stream sources (Lists, arrays, IO streams...) shouldn't change during streaming.
 - Non-concurrent sources will throw ConcurrentModificationException
 - It is possible to change concurrent sources (like ConcurrentLinkedList) during streaming
 - Concurrent collections uses Lock API in order to perform effective atomic locks when serving multiple threads

Streams - Lazily executed operations:

- Lazy execution means that when using these methods nothing really gets iterated...
 - Like creating a ‘view’ – which is basically a sub-stream
- Since each method results with a sub-stream – many operations can be chained without actually being evaluated:

```
stream.map(...).filter(...).limit(100)....
```

- Iteration starts when finalizing the stream via
 - Some basic operations like count(), findAny()...
 - Collectors (later)

Filtering streams

- Filters uses `Predicate<T>` as well but result with an updated stream
- The updated stream contains all the elements that passed the test
- `filter(Predicate <? super T>) : Stream<T>`

```
List<Person> people = new ArrayList<>();
//fill with 10,000 person instances
....
System.out.println("Females: "+people.stream().filter(p->p.getGender()==Gender.F).count());
System.out.println("Males: "+people.stream().filter(p->p.getGender()==Gender.M).count());
```

Output

Females: 5059
Males: 4941

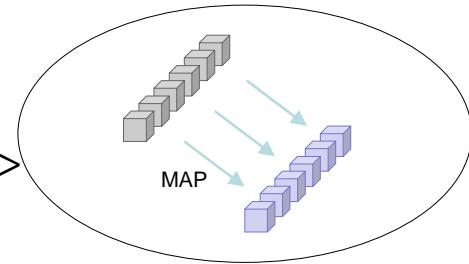
Mapping streams

- Mapping means to take an input and generate a related result out of it
- `map()` uses `Function<T,R>` with the `apply` method:
- `Function.apply(T) - returns R`
- Example – in order to calculate average age we need to map each person in the stream to its age collected in a new stream. Then, having an integers stream, we may calculate average

Java Streams API

Mapping streams

- So, Function.apply(T) - returns R
- Means that a Stream<T> can be mapped to Stream<R>
- Results are in the same order as T
- Streams API provides the following Mappers:
 - mapToDouble – apply(T) : double, produces DoubleStream
 - mapToInt – apply(T) : int, produces IntStream
 - mapToLong – apply(T) : long, produces LongStream
 - map – apply(T) : R produces Stream<R>
- Double, Int & Long Streams got additional collective methods:
 - average() – results in OptionalDouble/Int/Long
 - min(), max() – returns Optional<Double>/<Integer>/<Long>
 - boxed() – auto-box all primitives with their wrapper class
 - sum()



Java Streams API

Mapping streams

- Let's calculate average age

```
System.out.println(people.stream().mapToDouble(p -> p.getAge()).average().getAsDouble());
```

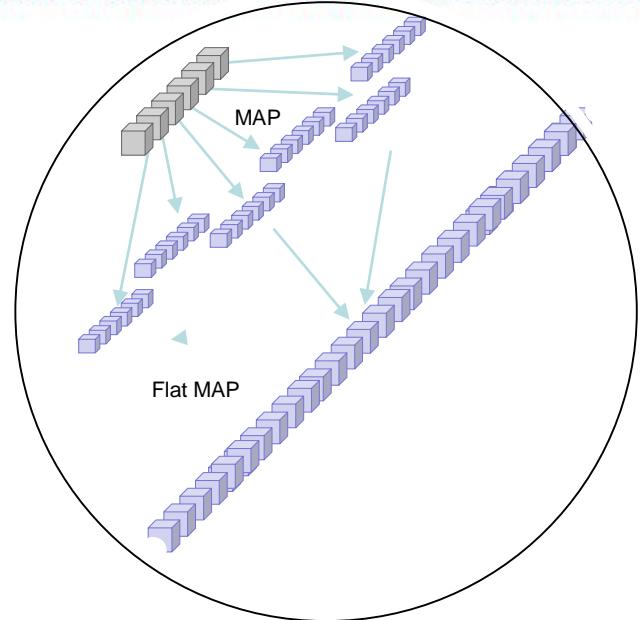
- Now, let's print the highest age value among the males in the list:

```
System.out.println(people.stream().filter(p -> p.getGender() == Gender.M)
    .mapToInt(p -> p.getAge()).max().getAsInt());
```

Java Streams API

Mapping streams

- flatMap() uses Function<T,Stream<R>> with the apply method:
- Function.apply(T) - returns Stream<R>
- Each element result is flattened into one big Stream<R>
- Streams API provides the following Mappers:
 - FlatMapToDouble – apply(T) : double, produces DoubleStream
 - FlatMapToInt – apply(T) : int, produces IntStream
 - FlatMapToLong – apply(T) : long, produces LongStream
 - FlatMap – apply(T) : R produces Stream<R>
- Double, Int & Long Flat Streams got same additional collective methods



Mapping streams

- Here we gather people phone numbers
- We assume each person holds a String array of phones available via `getPhones()` method
- We end up with a phone (String) stream holding all phone numbers from all people

```
...people.stream().flatMap(p -> Stream<String>.of(p.getPhones()))...
```

For Each with streams

- For each means to perform operation for each element in the stream
- forEach() takes a Consumer<T> and returns void
- This means forEach() is EAGERLY executed
- forEachOrdered() executes on each element in its original order in the stream
- Here, we let all the babies (age < 2) introduce themselves:

```
people.stream().filter(p->p.getAge()<2).forEach(p->System.out.println("Hi I'm "+p.getName()));
```

Peek with streams

- Peek acts just like forEach(..) – but with one major difference:
- peek() takes a Consumer<T> and returns a stream with the SAME elements
- Consumer code is executed on each element on its way to the new stream
- This is a big difference since peek() performs LAZILY

- Here, we call a method on each person just before filtering...

```
people.stream().peek(p->System.out.println(p.sayHello()))...filter(...
```

Reduce with streams

- Merges elements into single result
- Reduce() accepts:
 - T – which is the ‘merged’ value, starting with the first element
 - BiFunction<T,U,R> - U is current value to merge into T, R is the result of the merge
- This method performs EAGERLY
- Here, we sum letters of all person names

```
Stream<Integer>nameLengthStream = people.stream().map (p->p.getName().length());  
Optional<Integer> sum = lengthStream.reduce((x, y) -> x + y); sum.ifPresent(System.out::println);
```

Streams Collectors

- Collector generates a concrete result out of a stream
- Collectors utility class factors Collector of various types:
 - Simple collection collectors
 - ‘Single Result’ collectors
 - Manipulated collection collectors

Streams Collectors

- Simple collection collectors simply returns stream elements in a new Collection
- `toList()` – returns a List collector
- `toSet()` – returns a Set collector

```
List<Person> babies= people.stream().filter(p->p.getAge()<2).collect(Collectors.toList());  
System.out.println(babies.size());
```

Streams Collectors

- Simple collection collectors simply returns stream elements in a new Collection
- toMap() – returns a Map collector
- In this case each element is split to its logical key and value.
- Keys must be unique !
- Therefore, two mapping Function<T> must be provided - for both Key and Value
- toMap(Function<? Super T, ? extends K >, Function<? Super T, ? extends V >)
- Here, we generate a Map collection which holds baby ID as a key and its address as value:

```
Map<String,Address> babyAddresses=people.stream().filter(p->p.getAge()<2)
                                              .collect(Collectors.toMap(p->p.getID(), p-
>p.getAddress()));
```

Streams Collectors

- Creating ‘Single Result’ Collectors
- ‘Single Result’ collectors
 - Actually reduces the stream
 - Executes Function on each element and than fuses the outcome into single result
 - Counting collector is the simplest – simply counts elements

Streams Collectors

- Creating ‘Single Result’ Collectors – occurs when resulting in Collector sum value
 - All these methods return Collector.sum() which is the calculated result wrapped in:
 - averagingDouble (ToDoubleFunction< ? super T >) – results in Double value
 - averagingInt (ToIntFunction< ? super T >) – results in Integer value
 - averagingLong (ToLongFunction< ? super T >) – results in Long value
 - summingInt(), summingLong(), summingDouble()...
 - In order to simply count elements:
 - counting() – results in long value which is the element count
- In this example we calculate the average age of all young persons (age<18)

```
System.out.println(people.stream().filter(p->p.getAge()<18).  
collect(Collectors.averagingDouble(p->p.getAge())).doubleValue());
```

Streams Collectors

- Creating ‘Single Result’ Collectors – occurs when resulting in Collector sum value
 - When summarizing we get much more detailed information
 - Result is not *Collector.sum()* simple value – but *SummaryStatistics* instead
 - *summarizingDouble (...)* - returns DoubleSummaryStatistics
 - *summarizingInt (...)* - returns IntegerSummaryStatistics
 - *summarizingLong (...)* - returns LongSummaryStatistics
- SummaryStatistics methods:
 - *accept(T)* - records new value to the statistics
 - *getAverage()*
 - *getCount()*
 - *getMax()*
 - *getMin()*
 - *getSum()*

Streams Collectors

- Creating Manipulated Collection Collectors
- These collectors performs aggressive operation while collecting elements into collections:
 - Grouping by
 - Grouping by concurrent – works with *ConcurrentMaps*
 - Partitioning by
 - Joining

Streams Collectors

- Grouping by means generating a Map out of a Stream<T>
- A Function<T,K> calculates the key from each element
- Elements with identical keys are grouped into List / Set / Map Collectors
- Eventually, each Key is paired with a list of grouped elements - List<T>
- In this example we filter all persons which are 120 years old and group them by city:

```
Map<City, List<Person>> groups = people.stream().filter(p->p.getAge()==120)
                                              .collect(Collectors.groupingBy(p-
>p.getAddress().getCity(),Collectors.toList()));
System.out.println(groups);
```

Streams Collectors

- As mentioned, grouping by generates a Map in which a key is paired with a list of grouped elements - List<T>
- We may group by assigning ‘single result’ collectors as well:
- In this example we filter all persons which are 120 years old and count how many are living in each city:

```
Map<City, Long> groups = people.stream().filter(p->p.getAge()==120)
    .collect(Collectors.groupingBy(p-
    >p.getAddress().getCity(),Collectors.counting()));
System.out.println(groups);
```

Output:

```
{F=12, H=15, E=9, A=14, D=16, B=8, G=10, C=8}
```

Streams Collectors

- More group by examples:
- Showing how many persons live in each city:

```
Map<City, Long> groups = people.stream().collect(  
    Collectors.groupingBy(p->p.getAddress().getCity(), Collectors.counting()));  
System.out.println(groups);
```

Output:
{A=1321, B=1272, C=1193, D=1257, E=1249,
F=1204, G=1275, H=1229}

- Here we show average ages by gender:

```
Map<Gender, Double> groups = people.parallelStream().collect(  
    Collectors.groupingBy(Person::getGender,  
    Collectors.averagingInt(Person::getAge)));  
System.out.println(groups);
```

Output:
{F=60.81089743589744, M=55.755711775043935}

Streams Collectors

- Partitioning means creating a true/false key Map from a given stream
- `partitioningBy(Predicate<? super T>)` results with a `Map<Boolean,List<T>>`
- False key – holds a `List<T>` with elements that failed to pass the test
- True key – holds a `List<T>` with elements that passes the test

- Here we divide persons from city ‘A’ into 2 groups:
 - false - younger than 60
 - true – all the rest:

```
Map<Boolean,List<Person>> part=people.stream().filter(p -> p.getAddress().getCity().equals(City.A))  
                                .collect(Collectors.partitioningBy(p->p.getAge()>60));
```

Streams Collectors

- Concatenates the input elements
- Elements must be of type String
- Collectors.joining()
- A delimiter can be placed between each element: Collection.joining(String delimiter)
- Example of creating a long String with all person names :

```
System.out.println(people.stream().map(Person::getName).collect(Collectors.joining()));
```

Output:

```
!s!!o!!g!!e!!t!!n!!b!!g!!g!!j!!d!!r!!j!!i!!n!!b!!t!!b!!o!!b!!t!!e!!j!....
```

Reducing Streams

- Reducing means to fuse stream elements into single result
- reduce() accepts reduced value U, and current element T
- Reduce results is in fact an updated U value
- Updated U value is delegated to next element in the stream and so on..
- When reducing is started with initial value, result is U
- When reducing is started without it – result is Optional<U> - for null value outcome
- Mapping all person names and reducing into collection made of 2nd letter in each:
 - a is current reduced value
 - b is current element in the stream
 - b.name 2nd letter is added to, if not present already

```
System.out.println(people.stream().map(p ->
    p.getName())
    .reduce("", (a, b) -> {
        if(a.indexOf(b.charAt(1))==-1)
            a += b.charAt(1);
        return a;
    }));
}
```

Output:
{ajtbwhildxnrcspmuaekgvfoq}

Parallel Streams

- Fork-Join
 - All parallel stream share the same ForkJoinPool by default
 - The default ForkJoinPool uses a number of threads equals to the number of available processors -1
 - Fork-Join on stream includes:
 - Create a thread pool
 - Splitting the stream
 - Assigning each part to a thread consumed from the pool
 - Computing
 - Gathering results

Parallel Streams

- Pros & Cons
 - Parallelism is effective when:
 - There are limited number of threads &
 - Tasks are blocking for a long time
 - When processing intensive requests
 - We usually count on JVM to do the parallelism
 - Adding another business logic parallelism will cause slower execution
 - Parallel execution performance might vary dramatically due to other processing on the hosting machine

Parallel Streams

- When collecting parallel streams – use concurrent collections when possible

```
people.parallelStream().filter(p->p.getAge()>18).collect(Collectors.groupingByConcurrent(Person::getAge));
```

Parallel Streams

- Assigning dedicated pools to streams
 - Good for not sharing the default pool if too occupied

```
List<String> list = .....// A list of Strings
Stream<Integer> stream = list.parallelStream().map(String::length);

//we pause here and turn the stream into a Callable task.
//collect() eagerly starts the iteration and map Strings to their lengths:
Callable<List<Integer>> c = () -> stream.collect(toList());
ForkJoinTask task=ForkJoinTask.adapt(c);

//now, we create a NEW thread pool and assign the task to it
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
List<Integer> lengths = forkJoinPool.submit(task).get();
```



Thank You!