Advanced Java

Majrul Ansari
contactme@majrul.com
http://www.majrul.com

Overall Agenda

- ✓ Exploring commonly used features of Java like:
  - ✓ Multithreading
  - ✓ RMI
  - ✓ Java 5 & 6 improvements
  - ✓ Improved IO capabilities
  - ✓ Reflective API
  - ✓ Performance tuning
  - ✓ Garbage Collection
  - ✓ Profiling tools and utilities

Agenda for today's session (Day 1)

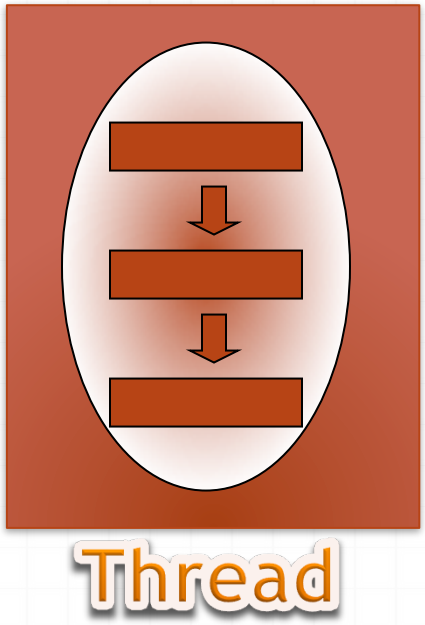
- ✓ **Multithreading**
  - ✓ Concurrency issues
  - ✓ Dead lock issues
  - ✓ Inter thread communication
  - ✓ Thread group
  - ✓ Synchronization
  - ✓ Volatile
  - ✓ Immutability
  - ✓ Defensive copies
  - ✓ Concurrent API
  - ✓ Non blocking algorithm
  - ✓ Future and Callable
  - ✓ Thread pool with Executor Framework
- ✓ **RMI**
  - ✓ Distributed Application development
  - ✓ RMI and it's working
  - ✓ Developing a simple client server application
  - ✓ Understanding RMIRegistry, UnicastRemoteObject and others

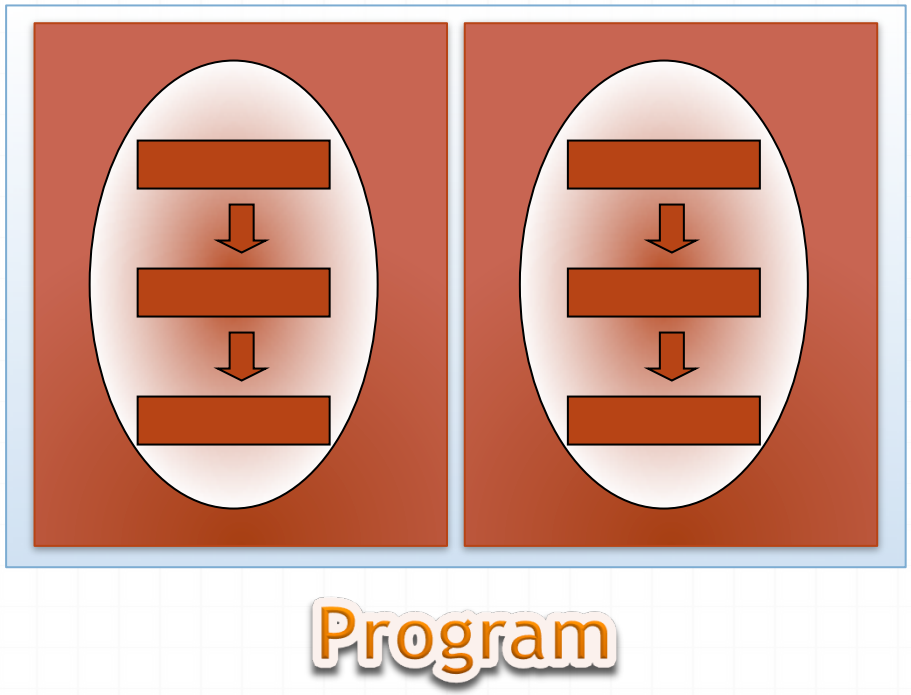
MultiThreading

Understanding different issues when writing Multithreaded code and exploring the new Concurrency API

What is a Thread?

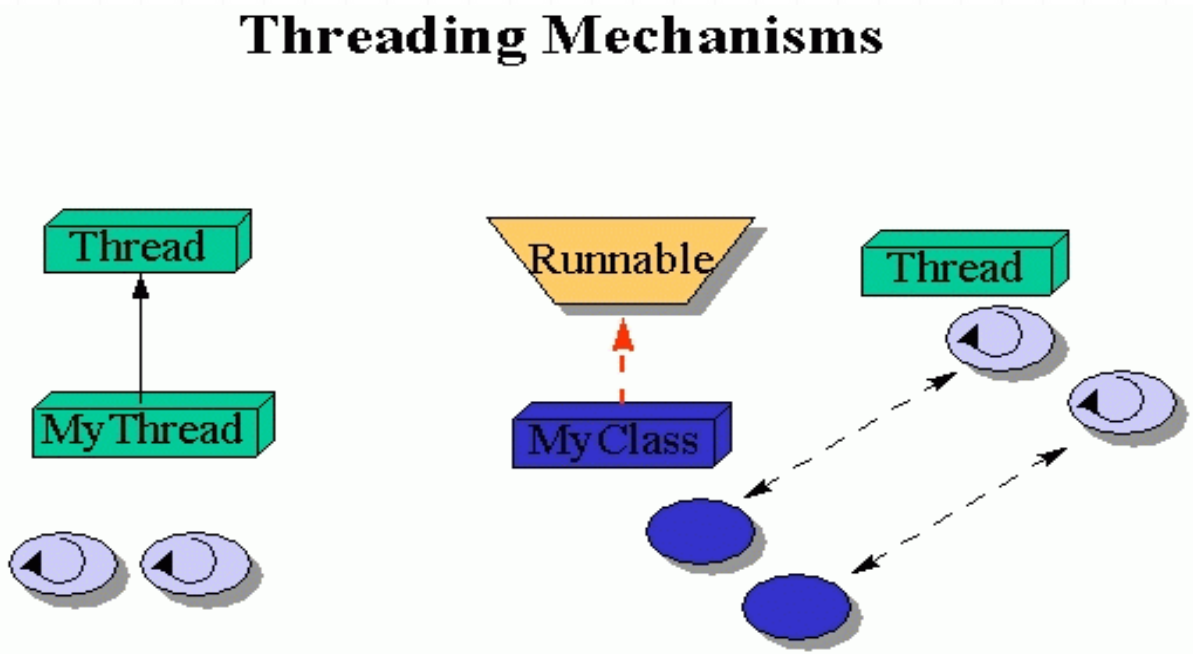
- A thread is a single sequential flow of control within a program
- We can also use multiple threads in a single program
  - Run at the same time and perform different tasks concurrently





How to create Java Threads

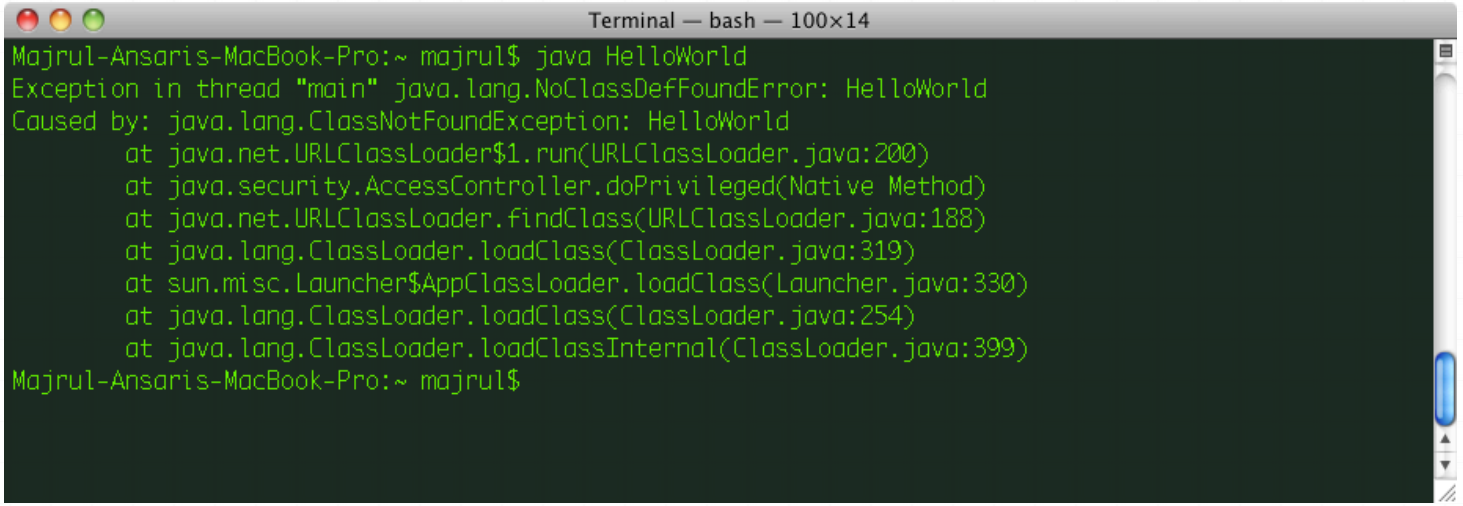
- There are two ways to create a Java thread:
  - Extend the java.lang.Thread class
  - Implement the java.lang.Runnable interface



Threads are lightweight processes

- ✓Threads run within the context of a full-blown process
- ✓Threads in a user task (process) can share code, data and system resources, and access them concurrently
- ✓Each Thread has a separate program counter, registers and a runtime stack
- ✓Each thread is allocated a predefined period of time to execute on a processor. When the time that is allocated for the thread expires, the thread's context is saved until its next turn on the processor, and the processor begins the execution of the next thread

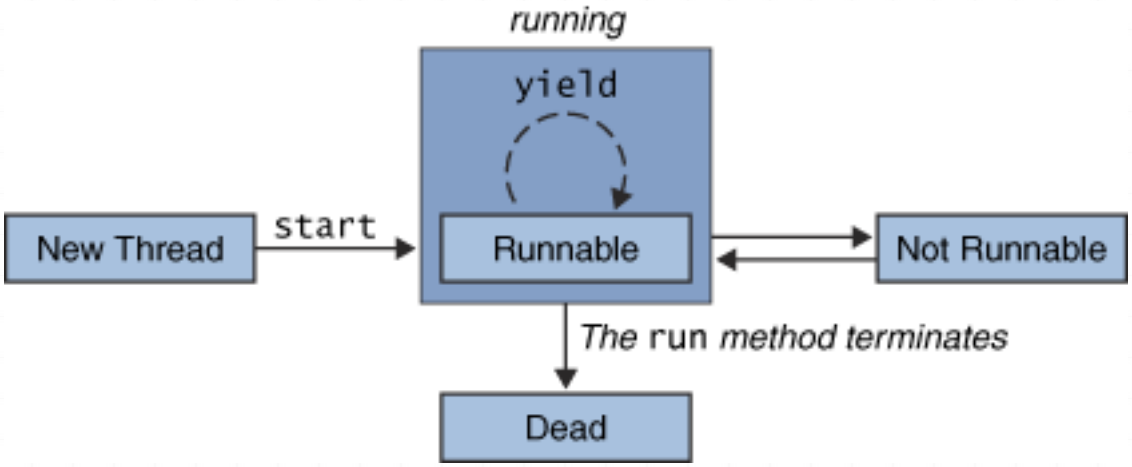
You must have seen this:



When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method

Thread lifecycle



Java 5 introduces a new getState() method in the Thread class so one we can use it for finding the current state of the thread. For ex:

Thread.State state = someThread.getState();

and the possible states are:
NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED

ThreadGroup

- Every Java thread is a member of a *thread group*. Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually
- If you create a new Thread without specifying its group in the constructor, the runtime system automatically places the new thread in the same group as the thread that created it
- By default the **main thread** belongs to a default thread group also by the same name **main**. So all user defined threads belong to the **main** thread group by default
- The parent of the main threadgroup is the **system** threadgroup whose members are those threads which are internally spawned by the JVM

Using ThreadGroup

```
public static void main(String[] args) {
     Thread myThread = new Thread();
     myThread.setPriority(1);
     myThread.start();
     ThreadGroup defaultGroup = myThread.getThreadGroup();
     defaultGroup.list();
```

Output:

```
java.lang.ThreadGroup[name=ma
in,maxpri=10]
    Thread[main,5,main]
    Thread[Thread-0,1,main]
```

```
ThreadGroup myGroup = new
ThreadGroup("MyGroup");
myThread = new Thread(myGroup, "MyThread");
myThread.setPriority(1);
myThread.start();
defaultGroup.list();
```

Output:

```
java.lang.ThreadGroup[name=main,maxpri=10]
   Thread[main,5,main]
   Thread[Thread-0,1,main]
java.lang.ThreadGroup[name=MyGroup,maxpri=10]
    Thread[MyThread,1,MyGroup]
```

Cont'd...

```
ThreadGroup sysGroup = defaultGroup.getParent();
sysGroup.list();
```

Output:

```
java.lang.ThreadGroup[name=system,maxpri=10]
    Thread[Reference Handler,10,system]
    Thread[Finalizer,8,system]
    Thread[Signal Dispatcher,9,system]
    java.lang.ThreadGroup[name=main,maxpri=10]
        Thread[main,5,main]
        Thread[Thread-0,1,main]
        java.lang.ThreadGroup[name=MyGroup,maxpri=10]
            Thread[MyThread,1,MyGroup]
```

Thread scheduling

- The most common thing to discuss about is how to control the execution of threads so that one thread doesn't holds the CPU for a long period of time and creating a situation where other threads are waiting for far too long to get a chance to get executed
- Execution of multiple threads on a single CPU in some order is called *scheduling*. The Java runtime environment supports a very simple, deterministic algorithm called *fixed-priority scheduling*. This algorithm schedules threads on the basis of their priority relative to other Runnable threads

Cont'd…

- If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run first. The chosen thread runs until one of the following conditions is true:
  - A higher-priority thread becomes runnable
  - The thread yields, or its run method exits
  - On systems that support time-slicing, the thread's time allotment has expired
- The Java runtime system's thread-scheduling algorithm is also preemptive. If at any time a thread with a higher priority than all other Runnable threads becomes Runnable, the runtime system chooses the new higher-priority thread for execution. The new thread is said to *preempt* the other threads

Cont'd…

```
public int tick = 1;
public void run() {
    while (tick < 10000000) {
        tick++;
    }
}
```

- The while loop in the above run method is in a tight loop. Once the scheduler chooses a thread with this thread body for execution, the thread never voluntarily relinquishes control of the CPU; it just continues to run until the while loop terminates naturally or until the thread is preempted by a high priority thread. This thread is called a *selfish thread*
- In some cases, having selfish threads doesn't cause any problems because a higher-priority thread preempts the selfish one. However, in other cases, threads with CPU-greedy run methods can take over the CPU and cause other threads to wait for a long time, even forever, before getting a chance to run

Relinquishing the CPU

- As you can understand, writing CPU-intensive code can have negative repercussions on other threads running in the same process. In general, try to write well behaved threads that voluntarily relinquish the CPU periodically and give other threads an opportunity to run
- Some of the methods which can be very helpful in allowing other threads to execute are:
  - yield()
    - A thread can voluntarily yield the CPU by calling the yield method, which gives other threads of the same priority a chance to run. If no equal-priority threads are Runnable, yield is ignored
  - sleep(milliseconds)
    - A thread can voluntarily go for a sleep so that any other thread which is waiting for the CPU can be executed

Race condition

- A *race condition* is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

Memory consistency errors

- *Memory consistency errors* occur when different threads have inconsistent views of what should be the same data. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them

  int counter = 0;

  - Suppose the counter field is shared between two threads, A and B. Suppose thread A increments counter:

    counter++;

  Then, shortly afterwards, thread B prints out counter:

    System.out.println(counter);

- If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a *happens-before* relationship between these two statements. To achieve *happens-before* relationship, one way of achieving the same is *synchronized* keyword

synchronized keyword

- By using synchronized keyword, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }
}
```

Cont’d…

- ✓ Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed
- ✓ When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a List called instances containing every instance of a class. You might be tempted to add the line **instances.add(this);** to your constructor. But then other threads can use instances to access the object before construction of the object is complete
- ✓ synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods

Important points on sync. keyword

- Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility
- Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock

Cont’d…

- ✓You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class

synchronized statements

- Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

```
public void handleRequest(…) {
    //reading user input
    //processing it
    synchronized(this) {
        hitCounter++;
    }
}
```

Another example

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Suppose, for example, class MsLunch has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks

Atomic access/volatile keyword

- In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete
- We have already seen that an increment expression, such as c++, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:
  - Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double)
  - Reads and writes are atomic for *all* variables declared volatile (*including* long and double variables)

volatile keyword

- Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.
- Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application

Liveness problems

- A concurrent application's ability to execute in a timely manner is known as its *liveness*
- *Deadlock* describes a situation where two or more threads are blocked forever, waiting for each other
- *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads.
- A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work

Immutable Objects

- An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.
- Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state
- Immutable objects can be useful in multi-threaded applications. Multiple threads can act on data represented by immutable objects without concern of the data being changed by other threads. Immutable objects are therefore considered to be more thread-safe than mutable objects

Mutable vs. Immutable

```
class Cart {
   private final List items;

   public Cart(List items) { this.items = items; }

   public List getItems() { return items; }
   public int total() { /* return sum of the prices
*/ }
}
```

```
class ImmutableCart {
   private final List items;

   public ImmutableCart(List items) {
     this.items =
Collections.unmodifiableList(new
ArrayList(items));
   }

   public List getItems() { return items; }
   public int total() { /* return sum of the prices
*/ }
 }
```

Depensive copying/cloning

- In an object is known to be immutable, it can be copied simply by making a copy of the reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a boost in execution speed.
- The reference copying technique is much more difficult to use for mutable object changes it, all other users of that reference will see the change. If this is not the intended effect, it can be difficult to notify the other users to have them respond correctly. In these situations, defensive copying (cloning) of the entire object rather than the reference is usually an easy but costly solution

Cloning in Java

- Unlike C++, objects in Java are always access indirectly through references. Objects are never created implicitly but instead are always passed or assigned by reference. Copying is usually performed by clone() method of Object class. This method by default performes a shallow copy and by overriding it we can provide custom cloning capability, like deep cloning

Collection classes

- ✓Since most of the collection classes are mutable and unsynchronized, there are many issues to deal with when using them in a shared mode. Only synchronizing the collection will not help since there is still an issue if one thread is reading and other thread is modifying it at the same time. This is where *Iterators* prove very handy. While reading the elements from a collection, if any other thread modifies the collection, immediately the iteration will stop and a *ConcurrentModificationException* will be raised. This which simply indicates that continuing further is of no use because we might end up showing an invalid result

Inter-thread communication

- ✓Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. This is where we can use *wait()*, *notify()*/*notifyAll()* pair to achieve effective communication
- ✓The following methods can be invoked from a *synchronized* block only
- ✓When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke notifyAll, informing all threads waiting on that lock that something important has happened and they can continue further now

Concurrency API

- When writing multithread applications, the issues that may create difficulties are related to data synchronization; these are the errors that make design harder, and such errors are hard to detect
- Built-in synchronization (methods and blocks) are fine for many lock-based applications, but they do have their own limitations, such as:
  - No way to back off from an attempt to acquire a lock that is already held, or to give up after waiting for a specified period of time, or to cancel a lock attempt after an interrupt
  - No way to alter the semantics of a lock, for example, with respect to reentrancy, read versus write protection, or fairness
  - No access control for synchronization. Any method can perform synchronized(obj) for any accessible object
  - Synchronization is done within methods and blocks, thus limiting use to strict block-structured locking. In other words, you cannot acquire a lock in one method and release it in another

Concurrency API Overview

- Concurrency API is divided into separate packages and classes catering to specific multithreading issues:
  - Atomic Variables: New set of classes for automatically manipulating single variables, which can be primitive types or references
  - Task Scheduling Framework: The Executor framework is for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies. Tasks are allowed to be executed within the submitting thread, in a single background thread (events in Swing), in a newly created thread, or in a thread pool
  - Concurrent Collections: New collection classes have been added, including Queue and BlockingQueue interfaces, as well as high-performance concurrent implementations of Map, List, and Queue
  - Synchronizers: General purpose synchronization classes that facilitate coordination between threads have been added including: semaphores, mutexes, barriers, latches, and exchangers.
  - Locks: Alternative to synchronized blocks and methods

Advantages of Concurrency API

- ✓ *Increased performance*:

  The implementations have been developed and tested by concurrency and performance experts, and therefore, they are faster and more scalable than typical implementations.

- ✓ *Increased reliability*:

  The low-level concurrency primitives (such as synchronized, wait(), and notify()) are difficult to use correctly...and errors are not easy to detect or debug. On the other hand, the concurrency utilities are standardized and extensively tested against deadlock, starvation, or race conditions

Non-blocking algorithm

- A non-blocking algorithm ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress; wait-free if there is also guaranteed per-thread progress
- Non-blocking algorithms are concurrent algorithms that derive their thread safety from not locks, but from low-level atomic hardware primitives such as *compare-and-swap* (CAS)
- Modern processors provide special instructions for atomically updating shared data that can detect interference from other threads, and the atomic package uses these instead of locking

Benefits

- The nonblocking version has several performance advantages over the lock-based version. It synchronizes at a finer level of granularity (an individual memory location) using a hardware primitive instead of the JVM locking code path, and losing threads can retry immediately rather than being suspended and rescheduled. The finer granularity reduces the chance that there will be contention, and the ability to retry without being descheduled reduces the cost of contention. Even with a few failed CAS operations, this approach is still likely to be faster than being rescheduled because of lock contention
- The most attractive property of nonblocking algorithms is the fact that if one thread fails (cache miss or worse seg fault) then other threads will not notice this failure and can continue on. However, when acquiring a lock, if the lock holding thread has some kind of OS failure every other thread waiting for the lock to be freed will be hit with the failure also

A bit more on CAS

- A CAS operation includes three operands -- a memory location (V), the expected old value (A), and a new value (B). The processor will atomically update the location to the new value if the value that is there matches the expected old value, otherwise it will do nothing
- The natural way to use CAS for synchronization is to read a value A from an address V, perform a multistep computation to derive a new value B, and then use CAS to change the value of V from A to B. The CAS succeeds if the value at V has not been changed in the meantime
- Instructions like CAS allow an algorithm to execute a *read-modify-write* sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation

Cont'd…

- An algorithm is said to be *wait-free* if every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads. By contrast, a *lock-free* algorithm requires only that *some* thread always make progress. (Another way of defining wait-free is that each thread is guaranteed to correctly compute its operations in a bounded number of its own steps, regardless of the actions, timing, interleaving, or speed of the other threads. This bound may be a function of the number of threads in the system; for example, if ten threads each execute the increment() operation once, in the worst case each thread will have to retry at most nine times before the increment is complete.)
- Substantial research has gone into wait-free and lock-free algorithms (also called *nonblocking algorithms*) over the past 15 years, and nonblocking algorithms have been discovered for many common data structures
- Nonblocking algorithms are used extensively at the operating system and JVM level for tasks such as thread and process scheduling. While they are more complicated to implement, they have a number of advantages over lock-based alternatives -- hazards like priority inversion and deadlock are avoided, contention is less expensive, and coordination occurs at a finer level of granularity, enabling a higher degree of parallelism.

java.util.concurrent.atomic package

- Until JDK 5.0, it was not possible to write wait-free, lock-free algorithms in the Java language without using native code. With the addition of the atomic variables classes in the java.util.concurrent.atomic package, that has changed
- This package provides classes that support lock-free thread-safe programming on single variables. The classes here extend the notion of volatile values, fields, and array elements.
- The classes in this package allow multiple operations to be treated atomically. As an example, a volatile integer cannot be used with the ++ operator because this operator contains multiple instructions. The AtomicInteger class has a method that allows the integer it holds to be incremented atomically without using synchronization. Atomic classes, however, can be used for more complex tasks, such as building code that requires no synchronization.
- Instances of classes AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference each provide access and updates to a single variable of the corresponding type. In addition, each class provides utility methods for that type, such as atomic increment methods.

Comparison

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

```
class AtomicCounter {
    private AtomicInteger c =
        new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

Cont'd…

- ✓The atomic variable classes can be thought of as a generalization of volatile variables, extending the concept of volatile variables to support atomic conditional compare-and-set updates. Reads and writes of atomic variables have the same memory semantics as read and write access to volatile variables.
- ✓While the AtomicCounter class might look superficially like the SynchronizedCounter example, the similarity is only superficial. Under the hood, operations on atomic variables get turned into the hardware primitives that the platform provides for concurrent access, such as CAS

Lock Objects

- ✓ Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the java.util.concurrent.locks package
- ✓ Lock objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a Lock object at a time. Lock objects also support a wait/ notify mechanism, through their associated Condition objects
- ✓ The biggest advantage of Lock objects over implicit locks is their ability to back out of an attempt to acquire a lock. The tryLock method backs out if the lock is not available immediately or before a timeout expires (if specified). The lockInterruptibly method backs out if another thread sends an interrupt before the lock is acquired
- ✓ So all this good set of features can prevent deadlocks very nicely as compared to synchronized keyword

Using ReentrantLock

```
public class ProductManager {

    private ReentrantLock lock = new ReentrantLock();

    public void updateProduct(Product product)
    throws ProductAccessException {
        try {
            if(lock.tryLock(500, TimeUnit.MILLISECONDS)) {
                //update product info in the database
                lock.unlock();
            }
            else throw new ProductAccessException("Resource busy");
        } catch (InterruptedException e) { }
    }
}
```

Instead of using synchronized keyword over here, we are simply using the Lock API to achieve the same but preventing other threads from blocking

Read/Write Locks

- When using a thread to read data from an object, you do not necessarily need to prevent another thread from reading data at the same time. So long as the threads are only reading and not changing data, there is no reason why they cannot read in parallel. The java.util.concurrent.locks package provides classes that implement this type of locking. The ReadWriteLock interface maintains a pair of associated locks, one for read-only and one for writing. The readLock() may be held simultaneously by multiple reader threads, so long as there are no writers. The writeLock() is exclusive. While in theory, it is clear that the use of reader/writer locks to increase concurrency leads to performance improvements over using a mutual exclusion lock. However, this performance improvement will only be fully realized on a multi-processor and the frequency that the data is read compared to being modified as well as the duration of the read and write operations

Using ReentrantReadWriteLock

```
public class RWDictionary {
    private final Map<String, Data> m = new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public Data get(String key) {
        r.lock();
        try { return m.get(key); }
        finally { r.unlock(); }
    }
    public String[] allKeys() {
        r.lock();
        try { return (String[])m.keySet().toArray(); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

Conditions using Lock

- ✓Conditions (also known as *condition queues* or *condition variables*) provide a means for one thread to suspend execution (to "wait") until notified by another thread that some state condition may now be true
- ✓Condition factors out the Object monitor methods into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. here a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods

Using Condition

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x)
   throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
```

```
 public Object take() throws
InterruptedException {
     lock.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length)
       takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
```

Executor framework

- The java.util.concurrent package defines three executor interfaces:
  - Executor, a simple interface that supports launching new tasks
  - ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself
  - ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks

Cont'd…

- The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace (new Thread(r)).start(); with e.execute(r);
- One of the most important feature introduced here is *worker threads* also called as *Thread Pooling*
- Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead

Traditional Webserver code

```
public class TraditionalWebServer {

    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(80);
        while(true) {
            Socket client = server.accept();
            //Runnable class
            RequestHandler requestHandler = new RequestHandler(client);
            Thread requestHandlerThread = new Thread(requestHandler);
            requestHandlerThread.start();
        }
    }
}
```

The problem in the above code are many. From the time taken to create a thread, plus the resources idle threads hold unnecessarily in the memory and then if our server accepts connections more than it can handle it will stop unexpectedly leaving system resources in an unacceptable state

Using ExecutorService

```
public class MuchBetterWebServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(80);
        ExecutorService pool = Executors.newFixedThreadPool(100);
        while(true) {
            Socket client = server.accept();
            //Runnable class
            RequestHandler requestHandler = new RequestHandler(client);
            pool.execute(requestHandler);
        }
    }
}
```

Executing threads in a pool as compared to creating new threads everytime increases performance plus with an upper bound to control the no of threads which can run at any moment of time prevents unexpected errors and gives us more control

Important methods

- ExecutorService as the name indicates provides methods to control the execution of threads
- An ExecutorService can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an ExecutorService. The shutdown() method will allow previously submitted tasks to execute before terminating while shutdownNow() method prevents waiting tasks from starting and attempts to stop currently executing tasks
- By combining the above two methods call with awaitTermination() method, we can wait for the ExectorService to shutdown before proceeding further

Callable objects

- Till now the only way to assign some task to a thread was by passing a Runnable object as a reference to the Thread constructor. Java 5 came up with one more way of submitting tasks to a Thread in the form of Callable object
- The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception

```
public interface Callable<V> {
    V call() throws Exception;
}
```

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

Future objects

- The simple need for multithreading in an application is to run asynchronous task whose result might not be available immediately. Since Runnable's run method doesn't return anything, this is where Callable is better suited for such tasks, for ex: querying the database etc…
- A very common question asked is how can we cancel a running task? This is where Future objects step in
- Future represents the lifecycle of a task and provides methods to test whether the task has been completed or been cancelled, retrieve it's result, and cancel the task
- FutureTask class can be used for wrapping Runnable objects to leverage all the benefits discussed above for existing code as well
- Executor supports both Runnable and Callable objects

Using Callable and Future

```
class LoadProductInfo implements Callable<ProductInfo> {

    @Override
    public ProductInfo call() throws Exception {
        //code to load product info from the db
        return new ProductInfo();
    }
}

ExecutorService exec = Executors.newCachedThreadPool();
Future<ProductInfo> future = exec.submit(new LoadProductInfo());
...
if(future.isDone()) {
    ProductInfo prodInfo = future.get();
}
else
    future.cancel(true);
```

By combining isDone(), isCancelled(), get(), get(timeout) we can efficiently manage concurrent tasks in our code

Using FutureTask

```
class LoadProductInfo implements Runnable {

     ProductInfo productInfo; //we need to populate this

     LoadProductInfo(ProductInfo productInfo) {
          this.productInfo = productInfo;
     }

     @Override
     public void run() {
          //code to load product info from the db
          productInfo.setXXX(…);
     }
}

     ExecutorService exec = Executors.newCachedThreadPool();
     ProductInfo productInfo = new ProductInfo();
     FutureTask<ProductInfo> task =
          new FutureTask<ProductInfo>(new LoadProductInfo(productInfo), productInfo);
     exec.execute(task);
     ...
     if(task.isDone()) {
          ProductInfo prodInfo = task.get();
     }
     else
          task.cancel(true);
```

Synchronizers

- **Semaphore**: A semaphore is a classic concurrency control construct. It is basically a lock with an attached counter. Similar to the Lock interface, it can be used to prevent access if the lock is granted. The Semaphore class keeps track of a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit. Note that no actual permit objects are used, the Semaphore maintains a count of the number available and acts accordingly. A semaphore with a counter of one can serve as a mutual exclusion lock.
- **Barrier**: This is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. The interface to the barrier is the CyclicBarrier class, called cyclic because it can be re-used after the waiting threads are released. This is useful for parallel programming.
- **CountDown Latch**: A latch is a condition starting out false, but once set true remains true forever. The CountDownLatch class serves as a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. This synchronization tool is similar to a barrier in the sense that it provides methods that allow threads to wait for a condition, but the difference is that the release condition is not the number of threads that are waiting. Instead, the threads are released when the specified count reaches zero. The initial count is specified in the constructor; the latch does not reset and later attempts to lower the count will not work.
- **Exchanger**: Allows two threads to exchange objects at a rendezvous point, and can be useful in pipeline designs. Each thread presents some object on entry to the exchange() method and receives the object presented by the other thread on return.

Using CountdownLatch

```
CountDownLatch latch = new CountDownLatch(2);

class Task1 implements Runnable {
     public void run() {
          //some task carried out
          latch.countDown();
          //continues
     }
}
class Task2 implements Runnable {
     public void run() {
          //some task carried out
          latch.countDown();
          //continues
     }
}
class Task3 implements Runnable {
     public void run() {
          //some task before entering wait
          try {
               latch.await(); //waiting for others to notify
          } catch (Exception e) { }
          //continues
     }
}
```

Using CyclicBarrier

```
CyclicBarrier barrier = new CyclicBarrier(2);

    class Task1 implements Runnable {
        public void run() {
            //some task
            try {
                barrier.await(); //waiting for other thread(s)
            } catch(Exception e) { }
            //continues
        }
    }

    private class Task2 implements Runnable {
        public void run() {
            //some task
            try {
                barrier.await(); //waiting for other thread(s)
            } catch(Exception e) { }
            //continues
        }
    }
```

Using Semaphore

```
private Semaphore semaphore = new Semaphore(5);

public Connection getConnection() {
    try {
        semaphore.acquire();
    } catch(Exception e) { }
    Connection conn = …;
    return conn;
}

public void releaseConnection(Connection conn) {
    try {
        conn.close();
    } catch(Exception e) { }
    semaphore.release();
}
```

Using Exchanger

```
class FillAndEmpty {
   Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
   DataBuffer initialEmptyBuffer = ... a made-up type
   DataBuffer initialFullBuffer = ...

   class FillingLoop implements Runnable {
     public void run() {
        //filling the bucket
       currentBuffer = exchanger.exchange(currentBuffer);
    }

   class EmptyingLoop implements Runnable {
     public void run() {
        /emptying the bucket
        currentBuffer = exchanger.exchange(currentBuffer);
    }

   void start() {
     new Thread(new FillingLoop()).start();
     new Thread(new EmptyingLoop()).start();
   }
 }
```

ThreadLocal

- ✓ThreadLocal class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread
- ✓In other words, if we need to provide a separate copy of some resource to each running thread, but if the same thread requests access to the resource again, another copy should not get created, then we can use ThreadLocal for such use cases

Not using ThreadLocal

```
class VeryBadDBManager {

    public Connection getConnection() {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            return DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Using ThreadLocal

```
class ExcellentDBManager {

        //anonymous inner class extending ThreadLocal
        private static ThreadLocal<Connection> connectionHandler =
        new ThreadLocal<Connection>() {
                protected Connection initialValue() {
                        try {
                                Class.forName("oracle.jdbc.driver.OracleDriver");
                                return DriverManager.getConnection(
                                        "jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");
                        }
                        catch (Exception e) {
                                throw new RuntimeException(e);
                        }
                };
        };
        public Connection getConnection() {
                return connectionHandler.get();
        }
}
```

Concurrent collections

- Every version of Java keeps on improving the Collection framework and in Java 5 the intention was to improve the concurrency capabilities of collections. Some of the collection classes are: ConcurrentHashMap, CopyOnWriteArrayList, CopyOnWriteArraySet, BlockingQueue, etc…
- The "Concurrent" prefix used with some classes in this package is a shorthand indicating several differences from similar "synchronized" classes. For example java.util.Hashtable and Collections.synchronizedMap(new HashMap()) are synchronized. But ConcurrentHashMap is "concurrent". A concurrent collection is thread-safe, but not governed by a single exclusion lock. In the particular case of ConcurrentHashMap, it safely permits any number of concurrent reads as well as a tunable number of concurrent writes. "Synchronized" classes can be useful when you need to prevent all access to a collection via a single lock, at the expense of poorer scalability. In other cases in which multiple threads are expected to access a common collection, "concurrent" versions are normally preferable

End of MultiThreading session

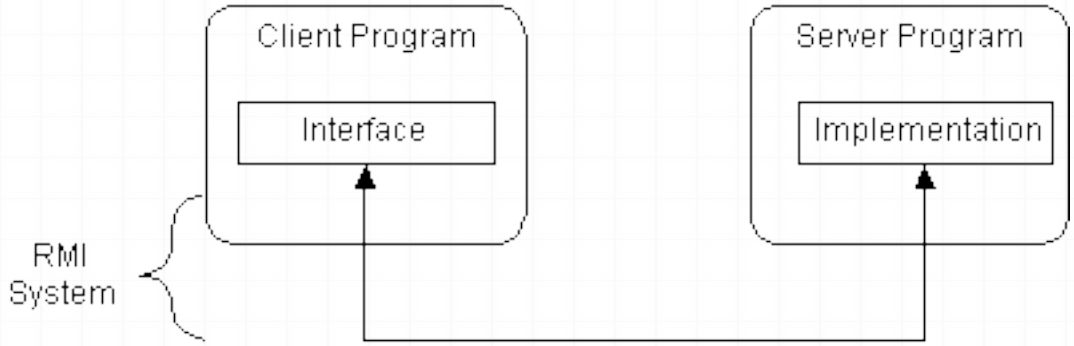
Java RMI

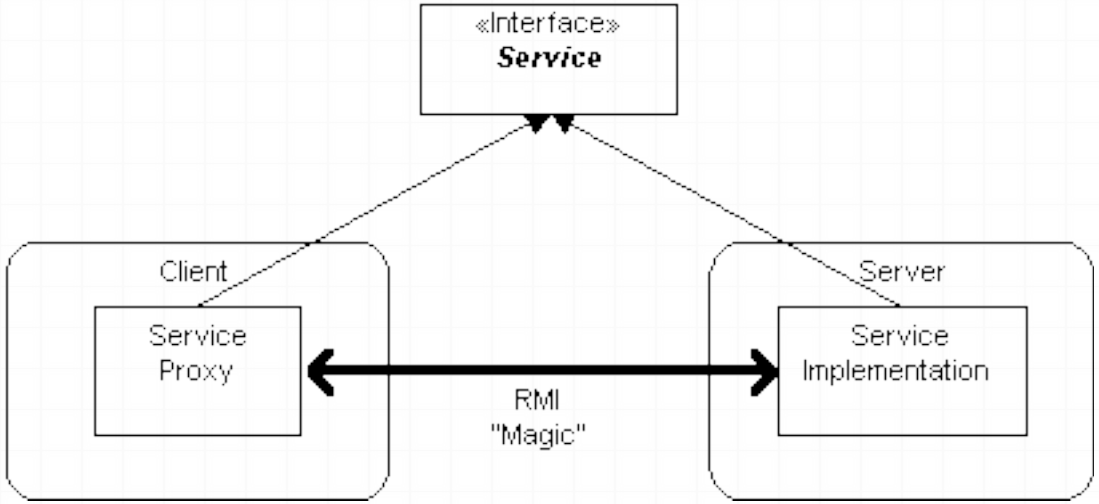
RMI also called as RPC in C/C++ allows code to be distributed across VM and access the same remotely

Java RMI

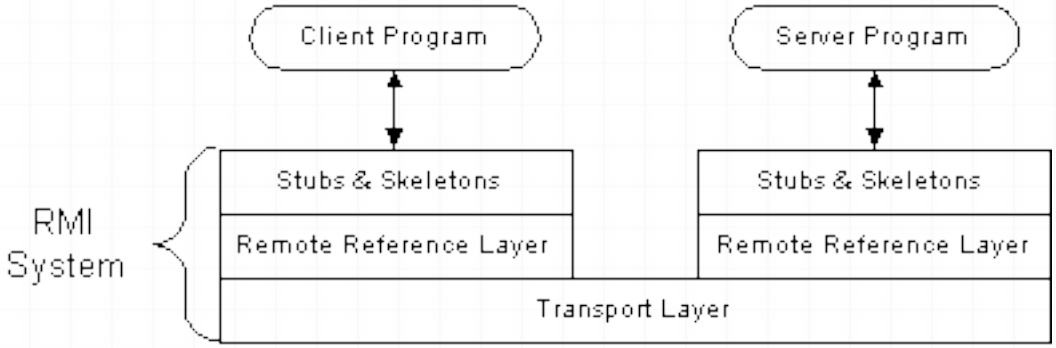
- The Java Remote Method Invocation (RMI) system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language
- RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*

The RMI system



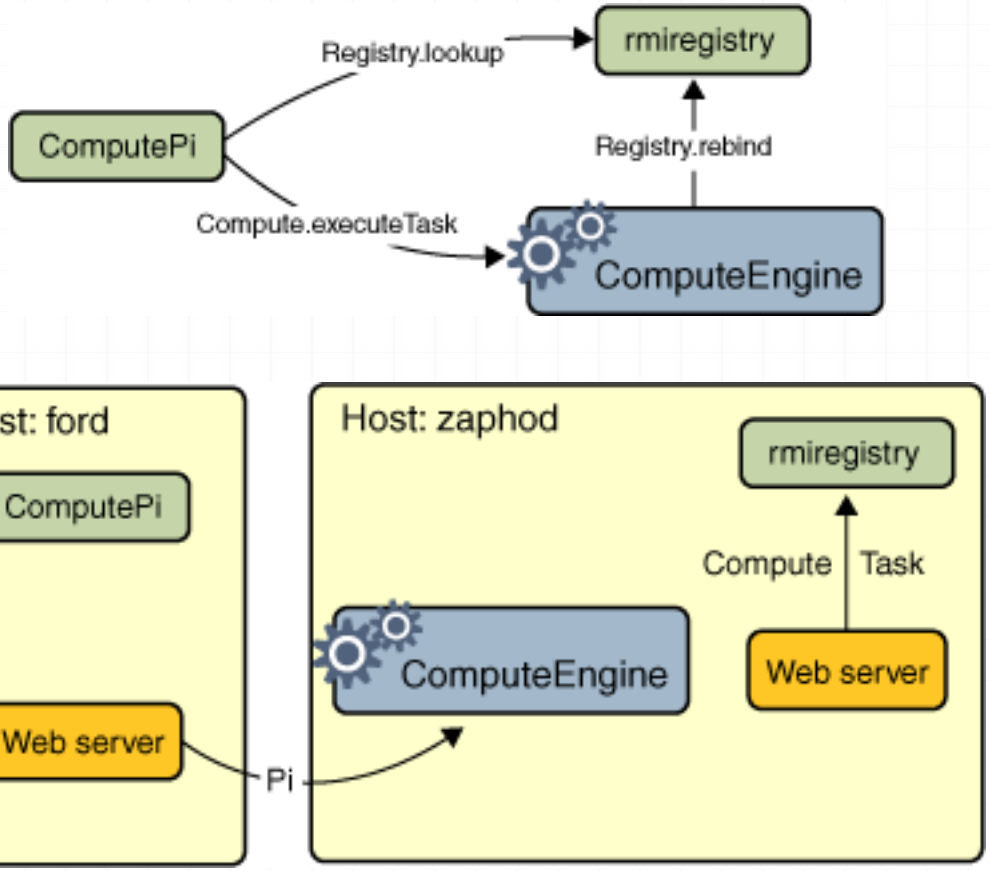


Stub and Skeleton



Stubs and Skeletons do all the hard work in any distributed architecture. Also called as proxies many a times. A very commonly used term is marshalling/unmarshalling and this is what stub/skeleton pair is responsible for. Marshalling/Unmarashalling means converting Java types like int, float, Serializable objects, into binary form so that they can be transmitted over the wire and vice-versaß

RMIRegistry



RMIRegistry is a service required for binding and lookup of remote references over the network. As long as the RMIRegistry is running, our service is running!

The common interface

```
public interface CardValidation extends Remote {
      public boolean isCardValid(String cardno)
                                    throws RemoteException;
}
```

Server side code

```
public class CardValidationService implements CardValidation {
    @Override
    public boolean isCardValid(String cardno) {
        //some business logic here
        return true;
    }
    public static void main(String[] args) throws Exception {
        CardValidation service = new CardValidationService();
        Registry registry = LocateRegistry.getRegistry();
        CardValidation stub =
            (CardValidation) UnicastRemoteObject.exportObject(service);
        registry.rebind("CCService", stub);
        System.out.println("Service started...");
    }
}
```

Client side code

```
public class CardValidationClient {

    public static void main(String[] args) throws Exception {
        Registry registry = LocateRegistry.getRegistry("localhost");
        CardValidation cardValidation =
                (CardValidation) registry.lookup("CCService");
        if(cardValidation.isCardValid("1234567890"))
            System.out.println(”Success!");
        else
            System.out.println(”Failure!");
    }
}
```

End of RMI session

End of Day 1

Agenda for today (Day 2)

- ✓New features of Java 5
- ✓Language improvements like Generics, Autoboxing, Annotations and others introduced in Java 5
- ✓Improvements in JDBC, XML Parsing and other core library changes
- ✓Support for I18N
- ✓Best practices
- ✓New features in Java IO
- ✓Reflection API
- ✓Creating Proxies
- ✓New features of Java 6

Beginning Java 5

Java 5 (1.5) introduced lots of new features which also effected the language and syntax of Java. When we talk about a new version, we generally ask for new Libraries, but this time it was drastic language level enhancements and new libraries as well

Java 5 improvements

- ✓ Language enhancements
  - ✓ Generics
  - ✓ Autoboxing/Unboxing
  - ✓ Enhanced for loop
  - ✓ Type safe enumerations
  - ✓ Varargs
  - ✓ Static imports
  - ✓ Metadata / Annotations
- ✓ API enhancements
  - ✓ JMX extensions for JVM Monitoring and Management
  - ✓ Concurrency API (we have already covered this one)
  - ✓ Utility APIs like Scanner, Formatter

Generics

- ✓ Generics eliminate the need of manual casting while accessing objects from a Collection.
- ✓ This is possible now since we can make the compiler aware about the usage of the Collection in our projects.
- ✓ This eliminates ClassCastException commonly occurring in Java applications
- ✓ Sample code without Generics:

```
ArrayList list = new ArrayList();
list.add(new Integer(100));
int value = ((Integer)list.get(0)).intValue();
```

- ✓ Sample code with Generics:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(100));
int value = list.get(0).intValue();
```

Cont'd…

- Let's see how does the new List and Iterator interface looks like:

```
interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

- So while using it, replace type parameter <E>, with the name of the Class for which the collection has to be instantiated, like Integer or MyClass.

Cont'd…

- ✓ The whole Collection API has been rewritten to support Generics
- ✓ Using Map without Generics:

```
HashMap map = new HashMap();
map.put( "majrul123" ,
            new
  Customer( "C123" ," Majrul" ," Ansari" ," Mumbai" ));
Customer cust = (Customer) map.get( "majrul123" );
```

- ✓ Using Map with Generics:

```
HashMap<String,Customer> map = new
                        HashMap<String, Customer>();
map.put( "majrul123" ,
            new
  Customer( "C123" ," Majrul" ," Ansari" ," Mumbai" ));
Customer cust = map.get( "majrul123" );
```

Using wildcards

- Suppose if we had to write some method which will process the collection, but we are not aware of the type of collection, then writing the following code will fail:

```
void process(Collection<Object> c) {
   //some code here
}
```

- Now the same code will look like this after using <?> wildcard:

```
void process(Collection<?> c) {
   //some code here
}
```

- The above code indicates the compiler, that this method accepts an unknown type

Cont'd…

```
List<Integer> li = new ArrayList<Integer>();
…
process(li); // no errors
Set<String> si = new HashSet<String>();
…
process(si);
```

- ✓Suppose we need to make sure that the parameter passed to the process method should be of a particular type, then:

```
void process(Collection<? extends Number> c) {
   …
}
```

Cont'd…

```
List<Integer> li = new ArrayList<Integer>();
process(li);
List<Long> li = new ArrayList<Long>();
process(li);
Collection<String> cs = new Vector<String>();
process(cs); //compile error
```

Writing our own Generic class

```
class Pair<F,S> {
    F first;
    S second;
    Pair(F f, S s) {
        first = f;
        second = s;
    }
    public void setFirst(F f) { first = f; }
    public F getFirst() { return first; }
    public void setSecond(S s) { second = s; }
    public S getSecond() { return second; }
    public static void main(String args[]) {
        Number n1 = new Integer(10);
        String s1 = new String("Java");
        Pair<Number,String> pair = new Pair<Number,String>(n1,s1);
    }
};
```

Autoboxing

- ✓ Java wrapper classes allow using primitives as objects. Commonly used with Collection API.
- ✓ So if we need to store an int, we need to wrap it up in an Integer class. This is called *boxing*.
- ✓ Similarly, if we need to extract int out of an Integer, we need to use intValue(). This is called *unboxing*.
- ✓ Autoboxing eliminates the overhead of boxing and unboxing from the developer.
- ✓ Without Autoboxing:

```
list.add(0, new Integer(42));
int val = (list.get(0)).intValue();
```

- ✓ Using Autoboxing:

```
list.add(0, new Integer(42));
int val = list.get(0);
```

Enhanced for loop

- ✓Iterating over collections is the most common thing developers write. For ex:

```
ArrayList<Integer> list = new ArrayList<Integer>();
For(Iterator I = list.iterator(); i.hasNext();) {
   Integer value = i.next();
   System.out.println(value);
}
```

- ✓Using the new for loop:

```
ArrayList<Integer> list = new ArrayList<Integer>();
for(Integer i : list) {
   System.out.println(i);
}
```

Enumerations

- ✓One of the missing feature required by Java developers for so long.
- ✓Without enums:

```
class Week {
   public static final int MONDAY = 0;
   public static final int TUESDAY = 1;
   public static final int WEDNESDAY = 2;
   public static final int THURSDAY = 3;
   ...
};
```

- ✓With enum:

```
enum Week { MONDAY, TUESDAY, WEDNESDAY, … }
```

Cont'd…

- ✓ Enums are full-fledged classes that:
  - ✓ Are type safe
  - ✓ Are robust
  - ✓ Define a namespace
  - ✓ Can have fields and methods
  - ✓ May implement arbitrary interfaces
  - ✓ Are Comparable and Serializable
- ✓ Iterating over enums:

```
enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY, SUNDAY;
}
for(Days day : Days.values())
    System.out.println(day);
```

Value-specific behavior

- This is a real good feature of enums. We can use the value of an enum for applying a specific behavior. For ex:

```
enum Operation {
    PLUS, MINUS, DIVIDE, MULTIPLY;

    public double eval(double x, double y) {
        switch(this) {
        case PLUS : return x+y;
        case MINUS : return x-y;
        case DIVIDE : return x/y;
        case MULTIPLY : return x*y;
        default : throw new Error("unknown operation");
        }
    }
}
```

Value-specific methods

- In this case, the value of an enum can be used to apply a certain behavior. For ex:

```
enum Operation2 {
    PLUS { double eval(double x, double y) { return x+y; }},
    MINUS { double eval(double x, double y) { return x-y; } };
    abstract double eval(double x, double y);
}
```

Varargs

- ✓Sometimes we need to define methods which accept multiple arguments. Now in such cases either we use collections or arrays and end up writing some extra code like this:

```
int sum(Integer[] numbers) {
      int mysum = 0;
      for(int i: numbers)
         mysum += i;
   return mysum;
}
…
int sum = sum(new Integer[] {12,13,20});
```

Cont'd…

- ✓Using Varargs:

```
public static int sum(Integer... numbers) {
        int mysum = 0;
       for(int i: numbers)
        mysum += i;
       return mysum;
   }
…
int sum = sum(10,20,30,40,50);
```

Static imports

- To avoid repetitive use of fully qualified names while accessing static members, we can use static imports. For ex:

```
import static java.lang.Math.*;

public class StaticImportTest1 {

    public static void main(String[] args) {
        System.out.println(sqrt(4));
    }
}
```

- Static imports can also be helpful for accessing enum variables.

Annotations / Metadata

- Annotations is not really something very new. Since Javadoc tags and XDoclet have already been there for adding metadata to Java code. JDK 5 takes annotations to the next level and standardizes the concept.
- Tiger provides six standard annotations:
  - java.lang.Override
  - java.lang.Deprecated
  - java.lang.Documented
  - java.lang.Inherited
  - java.lang.Retention
  - java.lang.Target

java.lang.Override

- This annotation if used forces the compiler to check if the method has been correctly overridden or not. This can be helpful since sometimes we think that we are overriding a method, but by mistake we are writing a new method.

```
class A {

    @Override
    public String toString(int x) {
        return "Hello";
    }
}
```

- The above code will generate a compile time error, since there is no toString(int) method in the superclass.

java.lang.Deprecated

- We use this annotation to mark a method that shouldn’ t be used anymore.

```
class A {

    @Deprecated
    public void test() {

    }

    public void testing() {
        test(); //compiler warning
    }
}
```

- The above code would result in a compiler warning.

Cont'd…

- java.lang.annotation.Documented
  - This annotation indicates that the annotation to which it is applied is to be documented by javadoc and similar tools. By default annotations are not documented by javadoc.
- java.lang.annotation.Inherited
  - If an annotation is used for the base class, then the same annotation would be applied to the subclasses also.
- java.lang.annotation.Retention
  - The Retention annotation takes a single parameter that determines the decorated annotation's availability. The values of this parameter are:
    - RetentionPolicy.SOURCE
    - RetentionPolicy.CLASS
    - RetentionPolicy.RUNTIME

java.lang.annotation.Target

- This annotation is used to indicate the type of program element (such as a class, method, or field) to which the declared annotation is applicable. If the Target annotation is not present on an annotation type declaration, then the declared type may be used on any program element. If the Target annotation is present, then the compiler will enforce the specified usage restriction. Legal values for the Target annotation are contained in the java.lang.annotation.ElementType enumeration. An annotation can be applied to any combination of a class, a method, a field, a package declaration, a constructor, a parameter, a local variable, and another annotation.

What is type erasure?

- So what happens, when we write the following code:

```
List<String> list = new ArrayList<String>();
String elem = list.get(0);
```

- The question is whether these new features effect the performance, and is there any difference in the bytecode generation, the answer is NO. Compilers use a simple concept called as *type erasure*, so finally the compiled code will look like:

```
List list = new ArrayList();
String elem = (String) list.get(0);
```

- The same concept applies to for-each loop, varargs, autoboxing and others

End of Java 5 language enhancements session

Reflection API

Reflection API can be used for achieving the highest level of polymorphism by dynamically accessing class attributes and behavior

Reflection API

- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. Reflection API can be used for different purposes:
  - Examining classes and objects
    - One can use Reflection API to collect metadata about an Object/ Class at the runtime. For ex: in eclipse, when we use the content-assist feature, it shows all possible options dynamically
  - Manipulating classes and objects
    - One can use Reflection API to dynamically instantiate objects, access fields, invoke methods, manipulate arrays, etc.. For ex: a WebServer like Tomcat uses Reflection API to dynamically execute the service() method of the corresponding servlet based on the URL submitted at the runtime

java.lang.Class class

- Class instance is the entry point in the Reflection API
- For every type of object, the Java virtual machine instantiates an immutable instance of java.lang.Class which provides methods to examine runtime properties of the object including its members and type information
- To get a hold of the Class of an object, we can use any of these different approach:
  - Class c = obj.getClass();
  - Class c = String.class;
  - Class c = Class.forName("java.lang.String");
  - Class c = int.class; //for using primitives in Reflection API

Collecting information

- After we get hold of a Class instance, we can now start introspecting the target class by using the different methods provided by Class class:
  - Method[] methods = c.getMethods();
  - Constructor[] constructors = c.getConstructors();
  - Field[] fields = c.getFields();
- Similarly many other methods are available to collect information about the package, parent class, implemented interfaces and annotations used

Invoking methods dynamically

```
class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}

Class calc = Calculator.class;
Object calcObj = calc.newInstance();
Method addMethod =
        calc.getMethod("add", int.class, int.class);
Object retValue = addMethod.invoke(calcObj, 10, 20);
System.out.println(retValue);
```

This is one of the most important feature of Reflection API, the power to invoke operations dynamically. We can invoke constructors, access fields, invoke methods, dynamically

Accessing private members

```
Class<Employee> empClass = Employee.class;
Employee empObj = empClass.newInstance();

Field empField = empClass.getDeclaredField("empno");
empField.setAccessible(true);
empField.setInt(empObj, 1001);

System.out.println(empField.getInt(empObj));
System.out.println(empObj.getEmpno());
```

One more powerful feature of Reflection API is to make it possible for accessing private members. In the above code, even though empno is a private field I am able to access it using Reflection API and change it's value. The only way to prevent someone from modifying private fields is make it final. Similarly we can invoke private methods also

Proxy objects

- Proxy pattern is one of the most popular design pattern used in our applications. Proxies acts as a simple drop in replacement and with the help of a factory class in between it is possible that we can completely hide the details about the proxy from the accessing code. Interfaces in Java enable loose coupling. So all these features combined together allows us to add extra capabilities to existing code without modifying the actual code
- Java Reflection API provides support for creating dynamic proxies. Dynamic proxy means a class created in memory with no physical .class file corresponding to it
- Each proxy instance has an associated invocation handler. When a method is invoked on a proxy instance, the method invocation is encoded and dispatched to the invoke method of its invocation handler

Returning the real object

```
public class EmployeeFactory {

    public static EmployeeService getEmployeeService() {
        EmployeeService empObj = new EmployeeServiceImpl();
        return empObj;
    }
}

EmployeeService empService =
    EmployeeFactory.getEmployeeService();
System.out.println(
    empService.applyForLeave(1001, "12/12/2011"));
```

Returning a proxy object

```
public class EmployeeFactory {

    public static EmployeeService getEmployeeService() {
        EmployeeService empObj = new EmployeeServiceImpl();
        EmployeeService empProxy =
            (EmployeeService) Proxy.newProxyInstance(
                EmployeeService.class.getClassLoader(),
                new Class[]{ EmployeeService.class },
                new MyInvocationHandler(empObj));
        return empProxy;
    }
}
```

```
public class MyInvocationHandler implements
InvocationHandler {

    private Object realObject;
    public MyInvocationHandler(Object realObject) {
        this.realObject = realObject;
    }
    @Override
    public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
        System.out.println("MyInvocationHandler");
        return method.invoke(realObject, args);
    }
}
```

End of Reflection API session

New features of Java IO

JDK 1.4 came up with a much faster and highly performant IO library called as NIO

New IO API (java.nio)

- The NIO API introduced in JDK 1.4 provides a completely new model of low-level IO. Unlike the original IO libraries in the java.io package, which is strongly stream oriented, the new IO API in the java.nio package is block oriented. This means that IO operations, wherever possible, are performed on large blocks of data in a single step, rather than on one byte or character at a time
- One of the most important feature which improves the performance of IO operations is called as direct buffers. Which means handling large files is now much more efficient as compared to the IO API when using NIO instead

Channels and Buffers

- ✓Channels and Buffers represent the two basic abstractions within the new IO API
- ✓Channels correspond roughly to input and output streams
- ✓The chunks of data that are written to and read from channels are contained in objects called buffers. A buffer is an array of data enclosed in an abstraction that makes reading from, and writing to, channels easy and convenient
- ✓Buffers are often large, reflecting the fact that the IO paradigm used in the NIO API is oriented towards transferring large amounts of data quickly

Cont'd…

- ✓Most of the existing IO classes in Java provide access to the underlying channel by invoking getChannel() method.

```
FileInputStream fin = new FileInputStream("file.txt");
FileChannel inc = fin.getChannel();
```

- ✓Before we can do any kind of IO on a channel, we need to have a buffer to do it with

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

- ✓Then using identical read()/write() methods available in the channel classes, we can start performing the rest of the IO operations using buffers

Without buffering. Only suffering!

```
public class Copy {

    public static void main(String[] args) {
        FileInputStream inFile = null;
        FileOutputStream outFile = null;
        try {
            inFile = new FileInputStream("ms.exe");//21 MB approx
            outFile = new FileOutputStream("newms.exe");
            int ch = 0;
            long ms1 = System.currentTimeMillis();
            while(true) {
                ch = inFile.read();
                if(ch == -1) break;
                outFile.write(ch);
            }
            long ms2 = System.currentTimeMillis();
            System.out.println(
                "File copied successfully in "+(ms2-ms1)+" ms");
        }
}
```

Output: File copied successfully in 299047 ms

With buffering. Less suffering!

```
try {
    inFile = new FileInputStream("ms.exe"); //21 MB approx
    outFile = new FileOutputStream("newms.exe");
    inBuffer = new BufferedInputStream(inFile, 1024*16);
    outBuffer = new BufferedOutputStream(outFile, 1024*16);
    int ch = 0;
    long ms1 = System.currentTimeMillis();
    while(true) {
        ch = inBuffer.read();
        if(ch == -1) break;
        outBuffer.write(ch);
    }
    long ms2 = System.currentTimeMillis();
    System.out.println(
        "File copied successfully in "+(ms2-ms1)+" ms");
}
```

Output: File copied successfully in 4494 ms

With NIO buffering. Nirvana!

```
try {
        inFile = new FileInputStream("ms.exe");
        outFile = new FileOutputStream("newms.exe");
        inChannel = inFile.getChannel();
        outChannel = outFile.getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(1024*16);
        long ms1 = System.currentTimeMillis();
        while(true) {
            int count = inChannel.read(buffer);
            if(count == -1) break;
            buffer.flip();
            outChannel.write(buffer);
            buffer.clear();
        }
        long ms2 = System.currentTimeMillis();
        System.out.println(
            "File copied successfully in "+(ms2-ms1)+" ms");

    }
```

Output: File copied successfully in 641 ms

How about speed of light?

```
try {
        inFile = new FileInputStream("ms.exe");
        outFile = new FileOutputStream("newms.exe");
        inChannel = inFile.getChannel();
        outChannel = outFile.getChannel();
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024*16);
        long ms1 = System.currentTimeMillis();
        while(true) {
            int count = inChannel.read(buffer);
            if(count == -1) break;
            buffer.flip();
            outChannel.write(buffer);
            buffer.clear();
        }
        long ms2 = System.currentTimeMillis();
        System.out.println(
            "File copied successfully in "+(ms2-ms1)+" ms");

    }
```

Output: File copied successfully in 324 ms

Handling huge files

- The problem with handling huge files is the memory footprint involved in performing any IO operation on it. If suppose we have a huge 100+ MB file in which we need to modify just one character, you can understand the rest of the problem. This is where java.nio.MappedByteBuffer steps in
- A MappedByteBuffer allows you to map a portion of a file into a memory buffer. The contents of the file are presented to the program as Buffer, which can be read from and written to like any other buffer. Changes made to the buffer are automatically propagated to the file by the underlying implementation of MappedByteBuffer
- The main advantage is speed. Data is retained in buffers at the OS level, rather than being copied into user space memory for each read() and out of user space memory for each write()

Cont'd…

- A MappedByteBuffer uses loading-on-demand, that is, it loads into memory only the data that is being accessed. Thus it is possible to create a 100 MB MappedByteBuffer that maps to a 100 MB file, and change an arbitrary byte in the file. This will cause the small portion around the changed byte to be loaded in the memory. The block's that aren't loaded don't take up any memory
- MappedByteBuffer gives us direct access to the operating system-level buffers. This means it is possible to access a file without any copying – the fastest access possible
- Because memory mapped files are tied so directly to the operating system, the design of the operating system can influence the way that the mapped buffers operate

Cont'd…

- ✓Under some operating systems, files are always read by mapping their data into blocks of memory, and these blocks are shared by all processes that access that file. An implementation of MappedByteBuffer can take advantage of this by using these low-level blocks directly. One side effect of this technique is that changes made to the contents of the MappedByteBuffer are seen instantly by other programs that are reading the file

Example

```
public static void main(String[] args) throws Exception {
    String filename = "ms.exe";
    int start = 100;
    int size = 100;
    long len = new File(filename).length();

    FileInputStream fin = new FileInputStream(filename);
    FileChannel finc = fin.getChannel();
    MappedByteBuffer mbb =
        finc.map(FileChannel.MapMode.READ_ONLY, start, size);

    long sum = 0;
    for(int i=0;i<size;i++)
        sum += mbb.get(i);

    System.out.println("sum : "+sum);
}
```

End of Java IO session

I18N

Internationalization, also referred to as Localization means the application is designed to support multiple different languages and regions

Internationalization

- Java provides full support for developing internationalized applications. Java provides full support for Unicode characters as well as a rich set of API for developing I18N applications
- The most important API classes are as follows:
  - java.util.Locale is an abstraction of a language and region
  - java.util.ResourceBundle allows externalization of strings
  - java.util.TimeZone & java.util.Calendar/GregorianCalendar provides date/time information specific to regions
  - java.text.NumberFormat & java.text.DateFormat provides methods for number & date formatting respectively

End of I18N session

Java 6

- JSE 6 has introduced lot's of new APIs for further enhancing Java application development
- No new language level enhancements were done in Java 6
- Some of the interesting new features of Java 6 are:
  - Compiler API
  - Support for Spring languages like Javascript
  - StAX(Streaming API for XML) parsing
  - JDBC 4.0
- Apart from this, other enhancements are:
  - JFC/Swing
  - Profiling
  - WebServices support
  - …

End of Java 6 features session

End of Day 2

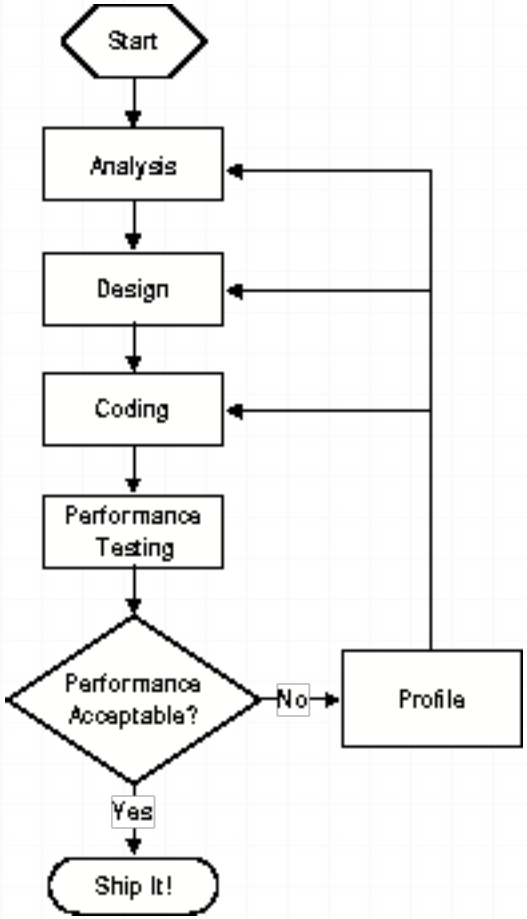
Agenda for today (Day 3)

- ✓Understanding JVM and different aspects of performance associated with it
- ✓Understanding JIT, GC, and how efficiently our Java programs run
- ✓Performance issues to deal with
- ✓Performance analysis tools like JConsole and VisualVM
- ✓Role of JMX in performance monitoring and management
- ✓Conclusion

Performance

- ✓When discussing about performance, there are different aspects to deal with and every aspect contributes to the final success of an application
- ✓Java applications run inside a managed environment of a JVM. But that doesn’t means, we developers have the freedom to write code which is bad and still expect things to go on smoothly. Expecting too much from the JVM is not a good decision. The initial stress should be on always writing optimal code which will perform well under different circumstances

A simple strategy



Performance improvements

- ✓Most of the new features we have seen so far improves the performance drastically. NIO, Concurrency API, Concurrent Collections, Non blocking algorithms are all must for writing high performance code
- ✓Also it is important to know when performance is must, then every single little thing will matter, like int and Integer and go on
- ✓Also since Collection classes are used very commonly, using the right one is essential to guarantee better performance
- ✓When using JDBC, it's important to use the latest versions of the JDBC driver so as to achieve better performance

A simple interesting example

- During the development of J2SE v. 1.3 the decision was made to rewrite the java.math package. This package contains classes for doing mathematical operations on very large numbers and contains many complex numerical algorithms. Previous versions of this code had been written almost entirely in C, but this made maintenance complex, so the code was rewritten using the Java programming language. However, it was critical that performance not suffer
- During the project it became clear that the JNI overhead for some operations was very high. Most lightweight operations became substantially faster when translated into Java code. In addition, removing the C code made the design so much cleaner that it was possible to change the implementation to use more efficient algorithms
- When the project was completed, benchmarks showed that most operations were much faster in the Java programming language version than they were in the C version. In many cases, operations were several hundred percent faster

Benchmarking and Testing

Benchmarking

- Benchmarking is the process of comparing operations in a way that produces quantitative results. Benchmarking plays a key role in ensuring that your software performs well
- A stopwatch can be a valuable benchmarking tool. It might seem a little silly to use a watch to help you tune high-performance software, but sometimes it's the best tool for the job. Obviously, you won't get millisecond accuracy with a stopwatch, but it's not always necessary to be that precise. For example, you might use a stopwatch to measure:
  - How long it takes to launch an application
  - How long it takes to open a large document
  - How long it takes to scroll through a very large table of data
  - How long it takes to execute a complex database query

Testing

- **What is load testing?**
- It is testing the server with realistic load to simulate a business peak hour. In real time, if the application has 100 concurrent users reading 1000 emails / hour, load testing is the simulation of the same scenario
- **What is stress testing?**
- Stress testing the stressing the system beyond the real time behavior. For example, test with 200 concurrent users reading 2000 emails / hour. Mostly this is done to simulate the breaking point of the system. The main objective is to determine the performance at unexpected load. One classic example is load on the Mobile Phone server due to SMS users. This is an unexpected load which may result in system to go down. Stress testing will help to determine application behaviour at these unexpected stress
- **What is Endurance testing?**
- Load and Stress testing is mostly done for 1 to 2 hours to simulate peak hour load. Endurance test is executed with average load for very long duration like 8 hours or more than that. The main objective of the test is to identify the performance issues over long run. One of the main reasons for endurance test is to identify issues like memory leak.
- Lets take the example above mentioned email application. Average load on this application is 100 concurrent users reading 5000 emails/ hour and it is for 24 hours a day. So the best endurance test scenario is to run the same for 24 hours minimum with 5000 transactions / hour. It should read 120000 emails / day

Memory footprint of an Application

How to know?

- ✓ Memory is occupied by two things:
  - ✓ Resources loaded by the VM process
  - ✓ Objects created by the Developer/API
- ✓ Program that uses too much memory can force the operating system to rely on virtual memory. Because virtual memory is many times slower than physical RAM, relying on virtual memory can result in slow performance and a poor user experience. The only easy to use option we had till now was java.lang.Runtime class providing methods like totalMemory() and freeMemory()
- ✓ JSE 5 introduces JMX as a way of monitoring JVM memory footprint. The java.lang.management introduces Mbeans form JVM montoring & management which we will discuss later
- ✓ GC helps in managing the Object space and freeing up unused objects, but that doesn't means you will never get *OutOfMemoryError*

Memory footprint of Classes

- When a source file is compiled with javac it produces a class file. Each method in the class is described by a set of bytecodes. When a class is loaded, the bytecodes for that class's methods must also be loaded so that when a method is executed the instructions are available to the virtual machine. These loaded bytecodes occupy space in RAM
- When a class is loaded by a virtual machine, two things happen:
  - The class file is loaded into RAM.
  - The contents of the file are parsed and reflective data structures are created inside the virtual machine to represent the class's methods and fields.
- Typically, the JVM creates structures to represent each method and field within the class. The size of these structures is totally dependent on the JVM implementation. However, inspection of a variety of JVM implementations shows that the size of the reflective structures can be substantial

Cont'd…

- ✓The key problem with classes, from a footprint perspective, is that you often load more than you need. Class loading is like a web-loading a single class often causes several others to be loaded, which in turn load more classes, and so on
- ✓The key is to try to load only what you use, and defer loading rarely-used features until they are needed

  java -verbose <MyMainClass>
- ✓The above option will help us figure out what are the classes getting loaded when we run our application

ClassLoading problem

```
public static Translator getTranslator(String fileType) {
    if (fileType.equals("doc")) {
       return new WordTranslator();
    } else if (fileType.equals("html")) {
       return new HTMLTranslator();
    } else if (fileType.equals("txt")) {
       return new PlainTranslator();
    } else if (fileType.equals("xml")) {
       return new XMLTranslator();
    } else {
       return new DefaultTranslator();
    }
}
```

If you enable –verbose option to run the above code, you will notice Translator, WordTranslator, HTMLTranslator, PlainTranslator, and XMLTranslator are all getting loaded in the VM

One solution

```
public static Translator getTranslator(String fileType) {
   try {
      if (fileType.equals("doc")) {
         return (Translator)Class.forName(
                        "WordTranslator").newInstance();
      } else if (fileType.equals("html")) {
         return (Translator)Class.forName(
                        "HTMLTranslator").newInstance();
      } else if (fileType.equals("txt")) {
         return (Translator)Class.forName(
                        "PlainTranslator").newInstance();
      } else if (fileType.equals("xml")) {
         return (Translator)Class.forName(
                        "XMLTranslator").newInstance();
      } else {
         return new DefaultTranslator();
      }
   } catch (Exception e) {
      return new DefaultTranslator();
   }
}
```

Using Reflection API, we can easily prevent eagerly loading of classes in the VM

GC

Garbage Collection

- Garbage collection (GC) is probably the most widely misunderstood feature of the Java platform. GC is typically advertised as removing all memory management responsibility from the application developer. This just isn't the case. On the other hand, some developers bend over backwards trying to please the collector, and often wind up doing much more work than is required. A solid understanding of the garbage collection model is essential to writing robust, high-performance software for the Java platform

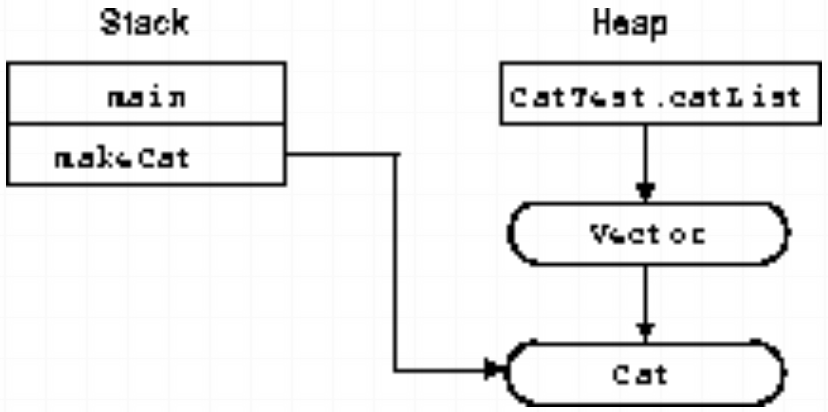
Object lifecycle

- In order to discuss garbage collection, it is first useful to examine the object lifecycle. An object typically goes through most of the following states between the time it is allocated and the time its resources are finally returned to the system for reuse
  - Created
  - In use (strongly reachable)
  - Invisible
  - Unreachable
  - Collected
  - Finalized
  - Deallocated

In Use

```
public class CatTest {
        static Vector catList = new Vector();
        static void makeCat() {
                Object cat = new Cat();
                catList.addElement(cat);
        }

        public static void main(String[] arg) {
                makeCat();
                // do more stuff
        }
}
```



Invisible

- An object is in the *invisible* state when there are no longer any strong references that are accessible to the program, even though there might still be references

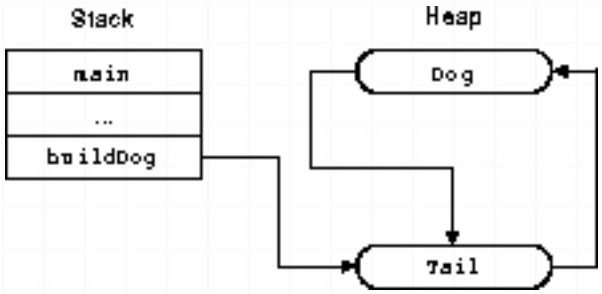
```
public void run() {
    try {
        Object foo = new Object();
        foo.doSomething();
    } catch (Exception e) {
        // whatever
    }
    while (true) { // do stuff } // loop forever
}
```

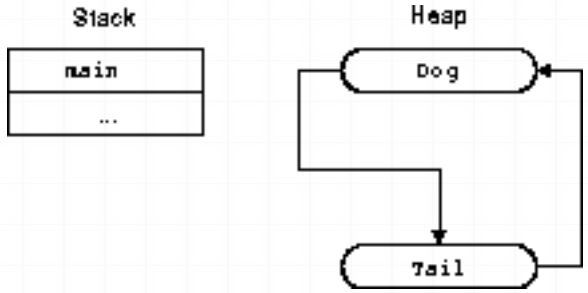
Unreachable

- An object enters an *unreachable* state when no more strong references to it exist. When an object is unreachable, it is a *candidate* for collection. Note the wording: Just because an object is a candidate for collection doesn't mean it will be immediately collected. The JVM is free to delay collection until there is an immediate need for the memory being consumed by the object
- It's important to note that not just any strong reference will hold an object in memory. These must be references that chain from a garbage collection root. GC roots are a special class of variable that includes
  - Temporary variables on the stack (of any thread)
  - Static variables (from any class)
  - Special references from JNI native code

Cont'd…

```
public void buidDog() {
   Dog newDog = new Dog();
   Tail newTail = new Tail();
   newDog.tail = newTail;
   newTail.dog = newDog;
}
```





At this point, the Dog and Tail both become unreachable from a root and are candidates for collection

Collected

- An object is in the *collected* state when the garbage collector has recognized an object as unreachable and readies it for final processing as a precursor to deallocation. If the object has a finalize method, then it is marked for finalization. If it does not have a finalizer then it moves straight to the finalized state
- If a class defines a finalizer, then any instance of that class must have the finalizer called prior to deallocation. This means that deallocation is delayed by the inclusion of a finalizer

Finalized

- An object is in the *finalized* state if it is still unreachable after its finalize method, if any, has been run. A finalized object is awaiting deallocation. Note that the VM implementation controls when the finalizer is run. The only thing that can be said for certain is that adding a finalizer will extend the lifetime of an object. This means that adding finalizers to objects that you intend to be short-lived is a bad idea. You are almost always better off doing your own cleanup instead of relying on a finalizer. Using a finalizer can also leave behind critical resources that won't be recovered for an indeterminate amount of time. If you are considering using a finalizer to ensure that important resources are freed in a timely manner, you might want to reconsider

Nice example on finalization

- One case where a finalize method delayed GC was discovered by the quality assurance (QA) team working on Swing. The QA team created a stress testing application that simulated user input by using a thread to send artificial events to the GUI. Running on one version of the toolkit, the application reported an OutOfMemoryError after just a few minutes of testing. The problem was finally traced back to the fact that the thread sending the events was running at a higher priority than the finalizer thread. The program ran out of memory because about 10,000 Graphics objects were held in the finalizer queue waiting for a chance to run their finalizers. It turned out that these Graphics objects were holding onto fairly substantial native resources. The problem was fixed by assuring that whenever Swing is done with a Graphics object, dispose is called to ensure that the native resources are freed as soon as possible

How GC works?

Stack & Heap

- ✓Memory is basically divided into stack and heap
- ✓Methods are loaded in the stack. Object references are kept in the stack of target method whereas the actual object is allocated in the heap
- ✓An object can be created using “new” keyword or using Reflection API’s “newInstance” method
- ✓Java HotSpot VM uses a generational GC as default
- ✓The heap is divided into two physical areas, which are referred to as generations, one young and one old

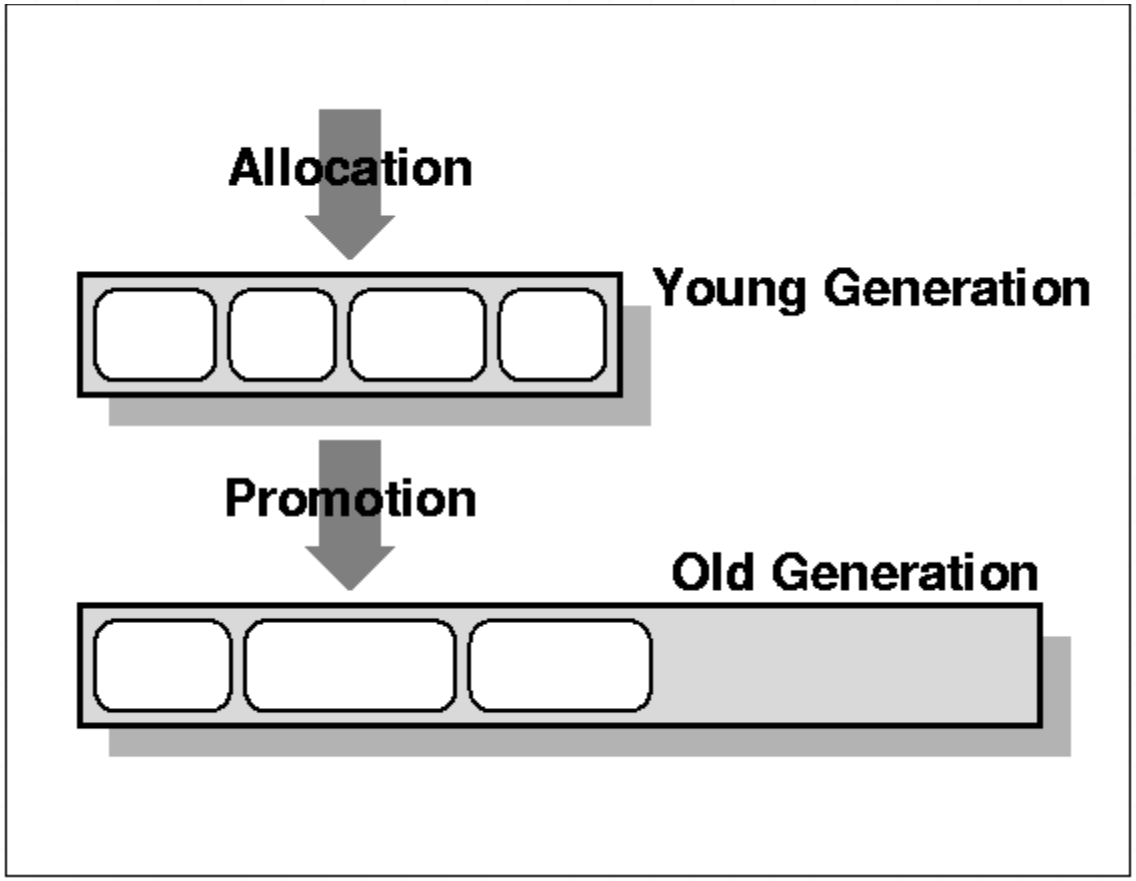
Young generation

- ✓Most newly allocated objects are allocated in the young generation, which is typically small and collected frequently. Since most objects in it are expected to die quickly, the number of objects that survive a young generation collection (also referred to as a minor collection) is expected to be low. In general, minor collections are very efficient because they concentrate on a space that is usually small and is likely to contain a lot of garbage objects.

Old Generation

- ✓Objects that are longer-lived are eventually promoted, or tenured, to the old generation. This generation is typically larger than the young generation and its occupancy grows more slowly. As a result, old generation collections (also referred to as major collections) are infrequent, but when they do occur they are quite lengthy.

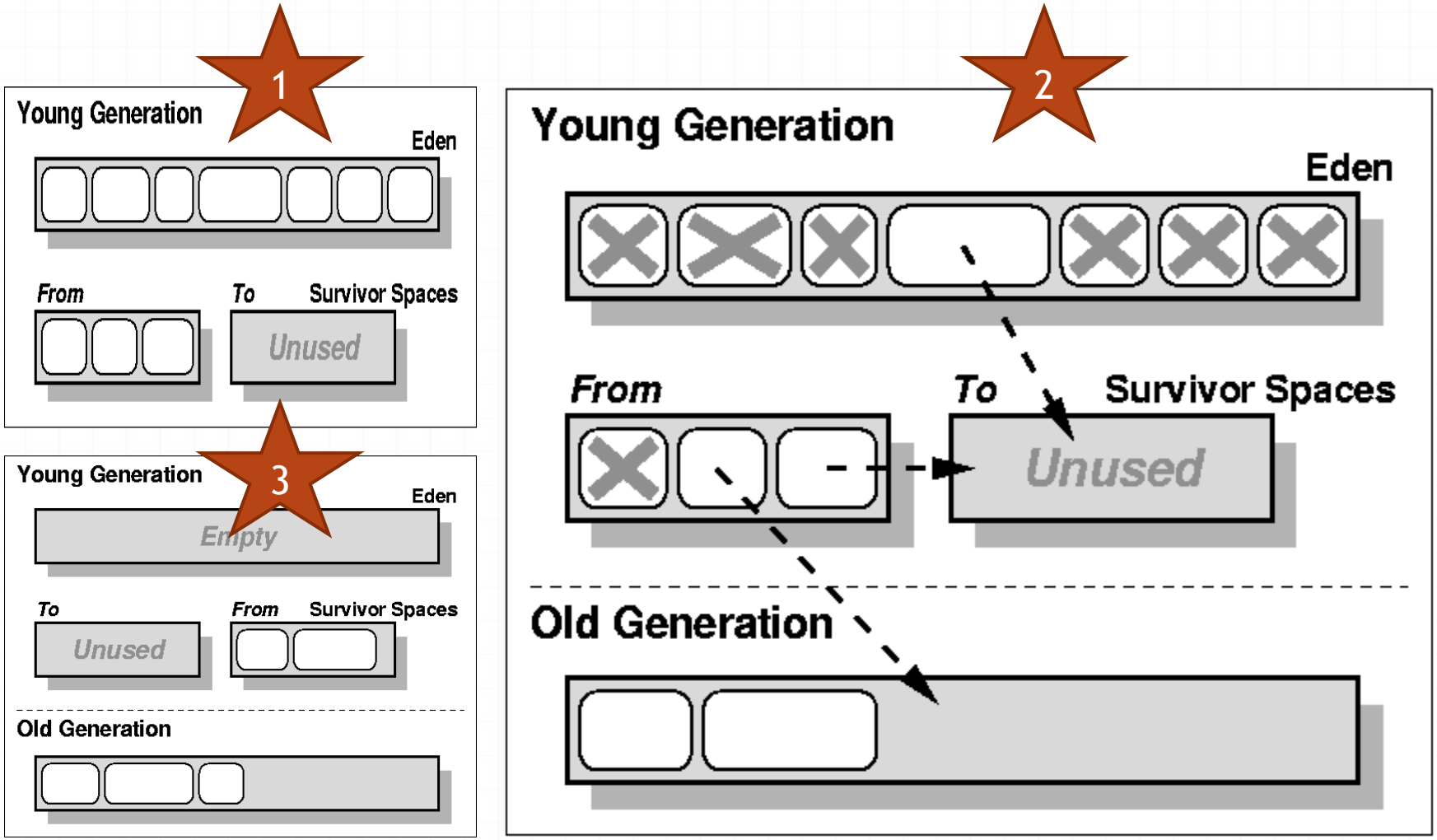
Young & Old Generation



The Young Generation

- The young generation is split into three areas:
  - The Eden: Most new objects are allocated here (large objects may be directly allocated into the old generation). The Eden is always empty after a minor collection.
  - The Two Survivor spaces: These hold objects that have survived at least one minor collection but have been given another chance to die before being promoted to the old generation.

Young Generation



How minor collection works

- Live objects in the Eden that survive the collection are copied to the unused survivor space. Live objects in the survivor space that is in use, which will be given another chance to die in the young generation, are also copied to the unused survivor space. Finally, live objects in the survivor space that is in use, which are deemed "old enough," are promoted to the old generation.

Cont’d…

- ✓At the end of the minor collection, the two survivor spaces swap roles. The Eden is entirely empty; only one survivor space is in use; and the occupancy of the old generation has grown slightly. Because live objects are copied during its operation, this type of collector is called a copying collector.
- ✓Because live objects are copied during its operation, this type of collector is called a copying collector.

Fast Allocation strategy

- The operation of the allocator is tightly coupled with the operation of the garbage collector. The collector has to record where in the heap the free space it reclaims is located. In turn, the allocator needs to discover where the free space in the heap is before it can re-use it to satisfy allocation requests. The copying collector that collects the young generation of the Java HotSpot VM has the advantage of always leaving the Eden empty, which allows allocations into the Eden to be very efficient by using the bump-the-pointer technique.
- According to this technique, the end of the last allocated object being tracked (usually called top) and when a new allocation request needs to be satisfied, the allocator needs only to check whether it will fit between top and the end of the Eden. If it does, top is bumped to the end of the newly allocated object.

About Generational GC

- A generational collector takes advantage of the fact that in most programs,the vast majority of objects (often greater than 95 percent) are very short lived (for example, they are used as temporary data structures). By segregating newly created objects into an object nursery, a generational collector can accomplish several things. First, because new objects are allocated contiguously in stack-like fashion in the object nursery, allocation becomes extremely fast, since it merely involves updating a single pointer and performing a single check for nursery overflow. Secondly, by the time the nursery overflows, most of the objects in the nursery are already dead, allowing the garbage collector to simply move the few surviving objects elsewhere, and avoid doing any reclamation work for dead objects in the nursery

Parallel Young Generation GC

- The single-threaded copying collector described above, while suitable for many deployments, could become a bottleneck to scaling in an application that is otherwise parallelized to take advantage of multiple processors. To take full advantage of all available CPUs on a multiprocessor machine, the Java HotSpot VM offers an optional multithreaded collector for the young generation, in which the tracing and copying of live objects is accomplished by multiple threads working in parallel. The implementation has been carefully tuned to balance the collection work between all available processors, allowing the collector to scale up to large numbers of processors. This reduces the pause times for collecting young space and maximizes garbage collection throughput. The parallel collector has been tested with systems containing more than 100 CPU's and 0.5 terabytes of heap. The parallel young generation collector is the default garbage collection algorithm used with the Server VM

Different GC's

- Garbage Collectors come in two types:
  - Serial Collector
  - Parallel Collector
- With the serial collector, both young and old collections are done serially (using a single CPU), in a stop-the-world fashion. That is, application execution is halted while collection is taking place. When using the Client VM, the Serial Collector is used by default
- When using the Server VM, the Parallel collector is used by default so that the GC can run on a different CPU at the same time without pausing the application for a long time

Cont'd...

- ✓Serial Collector
  - ✓Young Generation Collection using the Serial Collector
  - ✓Old Generation Collection using Serial Collector
- ✓Parallel Collector
  - ✓Young Generation Collection using Parallel Collector
  - ✓Old Generation Collection using Parallel Collector
- ✓Parallel Compacting Collector
  - ✓Young Generation Collection using Compacting Collector
  - ✓Old Generation Collection using Compacting Collector
- ✓Concurrent Mark-Sweep (CMS) Collector
  - ✓Young Generation Collection using CMS Collector
  - ✓Old Generation Collection using CMS Collector

Parallel Compacting Collector

- With the parallel compacting collector, the old and permanent generations are collected in a stop-the-world, mostly parallel fashion with sliding compaction. The collector utilizes three phases
  - Marking phase
  - Summary phase
  - Compaction phase

Marking phase

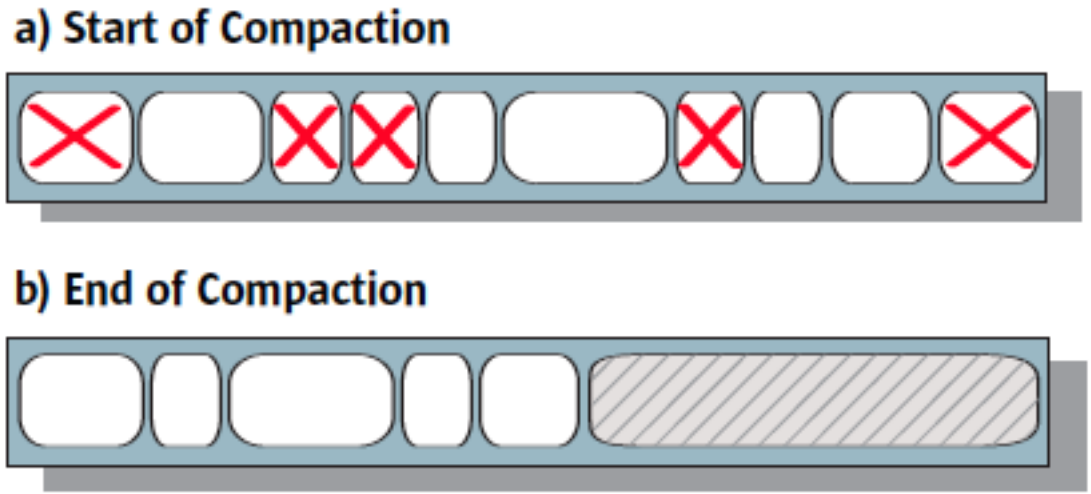
- In the *marking phase*, the initial set of live objects directly reachable from the application code is divided among garbage collection threads, and then all live objects are marked in parallel. As an object is identified as live, the data for the region it is in is updated with information about the size and location of the object

Summary phase

- The summary phase operates on regions, not objects. Due to compactions from previous collections, it is typical that some portion of the left side of each generation will be dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary phase does is examine the density of the regions, starting with the leftmost one, until it reaches a point where the space that could be recovered from a region and those to the right of it is worth the cost of compacting those regions. The regions to the left of that point are referred to as the dense prefix, and no objects are moved in those regions. The regions to the right of that point will be compacted, eliminating all dead space. The summary phase calculates and stores the new location of the first byte of live data for each compacted region. Note: The summary phase is currently implemented as a serial phase; parallelization is possible but not as important to performance as parallelization of the marking and compaction phases

Compaction phase

- In the compaction phase, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end



CMS Collector

- A collection cycle for the CMS collector starts with a short pause, called the initial mark, that identifies the initial set of live objects directly reachable from the application code. Then, during the concurrent marking phase, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called remark, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency

Cont'd…

- ✓At the end of the remark phase, all live objects in the heap are guaranteed to have been marked, so the subsequent concurrent sweep phase reclaims all the garbage that has been identified
- ✓Since some tasks, such as revisiting objects during the remark phase, increase the amount of work the collector has to do, its overhead increases as well. This is a typical trade-off for most collectors that attempt to reduce pause times
- ✓The CMS collector is the only collector that is non-compacting. That is, after it frees the space that was occupied by dead objects, it does not move the live objects to one end of the old generation

CMS

a) Start of Sweeping

b) End of Sweeping

Cont'd…

- This saves time, but since the free space is not contiguous, the collector can no longer use a simple pointer indicating the next free location into which the next object can be allocated. Instead, it now needs to employ free lists. That is, it creates some number of lists linking together unallocated regions of memory, and each time an object needs to be allocated, the appropriate list (based on the amount of memory needed) must be searched for a region large enough to hold the object. This also imposes extra overhead to young generation collections, as most allocations in the old generation occur when objects are promoted during young generation collections

HotSpot VM

Introduction

- ✓he Java HotSpot Virtual Machine is a core component of the Java SE platform. It implements the Java Virtual Machine Specification, and is delivered as a shared library in the Java Runtime Environment. As the Java bytecode execution engine, it provides Java runtime facilities, such as thread and object synchronization, on a variety of operating systems and architectures. It includes dynamic compilers that adaptively compile Java bytecodes into optimized machine instructions and efficiently manages the Java heap using garbage collectors, optimized for both low pause time and throughput. It provides data and information to profiling, monitoring and debugging tools and applications

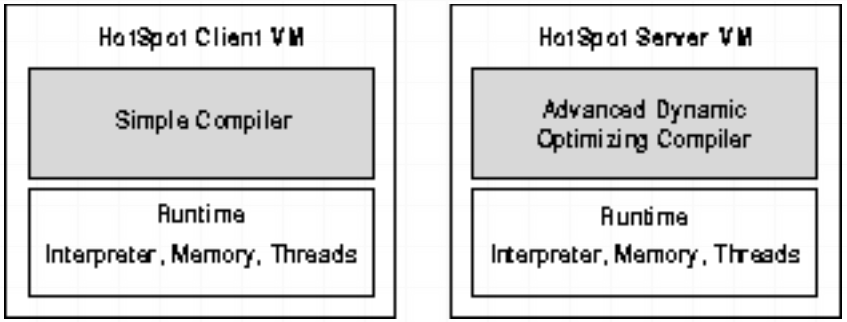
Cont’d…

- ✓HotSpot is an "ergonomic" JVM. Based upon the platform configuration, it will select a compiler, Java heap configuration, and garbage collector that produce good to excellent performance for most applications. Under special circumstances, however, specific tuning may be required to get the best possible performance. The resources collected here will help the reader understand and tune the Java HotSpot Virtual Machine

Flavors of HotSpot VM

- The JDK includes two flavors of the VM -- a client-side offering, and a VM tuned for server applications. These two solutions share the Java HotSpot runtime environment code base, but use different compilers that are suited to the distinctly unique performance characteristics of clients and servers. These differences include the compilation inlining policy and heap defaults
- The JDK contains both of the these systems in the distribution, so developers can choose which system they want by specifying -client or -server

Client vs. Server VM



The Client VM and the Server VM are very similar, and actually share a lot of code. The only part of the system that is different is the compiler. The Server VM contains a highly advanced adaptive compiler that supports many of the same types of optimizations performed by optimizing C++ compilers (as well as a few optimizations C++ compilers only wish they could do). The Client VM is much simpler. It doesn't try to perform many of the more complex optimizations performed by the compiler in the Server VM, but in exchange the Client VM requires less time to analyze and compile a particular piece of code. This means that the Client VM can start up faster, and requires less warm-up time to reach peak performance.

Cont'd…

- ✓The Server VM has been specially tuned to maximize peak operating speed. It is intended for executing long-running server applications, which need the fastest possible operating speed more than a fast start-up time or smaller runtime memory footprint
- ✓The Client VM compiler serves as an upgrade for both the Classic VM and the just-in-time (JIT) compilers used by previous versions of the JDK

Memory model of HotSpot VM

- In previous versions of the Java virtual machine, such as the Classic VM, indirect handles are used to represent object references. While this makes relocating objects easier during garbage collection, it represents a significant performance bottleneck, because accesses to the instance variables of Java programming language objects require two levels of indirection
- The Classic JVM represents an object as a pointer to a data structure called a **handle**, which contains a pointer to the instance data. This object layout results in an unnecessary memory reference for every object access

Cont’d…

- In the Java HotSpot VM, no handles are used by Java code. Object references are implemented as direct pointers. This provides C-speed access to instance variables. When an object is relocated during memory reclamation, the garbage collector is responsible for finding and updating all references to the object in place
- The Java HotSpot VM uses a two machine-word object header, as opposed to three words in the Classic VM. Since the average Java object size is small, this has a significant impact on space consumption -- saving approximately eight percent in heap size for typical applications. The first header word contains information such as the identity hash code and GC status information. The second is a reference to the object's class. Only arrays have a third header field, for the array size

HotSpot VM and Threads

- Per-thread method activation stacks are represented using the host operating system's stack and thread model. Both Java programming language methods and native methods share the same stack, allowing fast calls between the C and Java programming languages. Fully preemptive Java programming language threads are supported using the host operating system's thread scheduling mechanism
- A major advantage of using native OS threads and scheduling is the ability to take advantage of native OS multiprocessing support transparently. Because the Java HotSpot VM is designed to be insensitive to race conditions caused by preemption and/or multiprocessing while executing Java programming language code, the Java programming language threads will automatically take advantage of whatever scheduling and processor allocation policies the native OS provides

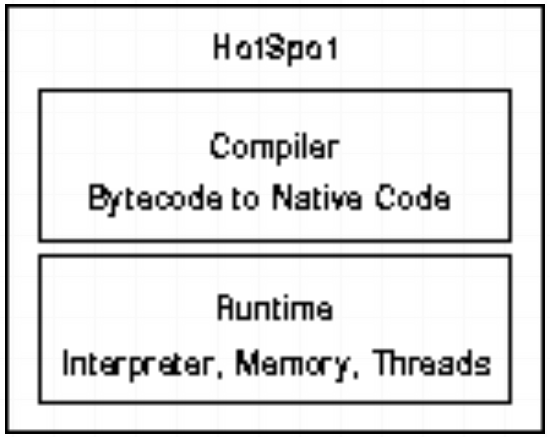
What is Class Data Sharing?

- The class data sharing feature is aimed at reducing application startup time and footprint. The installation process loads a set of classes from the system jar file into a private, internal representation, then dumps that representation to a "shared archive" file. During subsequent JVM invocations, the shared archive is memory-mapped in, saving the cost of loading those classes and allowing much of the JVM's metadata for these classes to be shared among multiple JVM processes.

JIT

Introduction

- There are two main parts to the HotSpot system: the runtime and the compiler. The runtime portion includes a bytecode interpreter, memory management and garbage collection functionality, and machinery for handling thread synchronization and other low-level tasks. The compiler's job is simply to translate bytecodes into native machine instructions, thus improving execution speed



Cont’d…

- ✓The HotSpot VM uses a combination of bytecode interpretation and dynamic compilation. When a class is firtst loaded, the JVM executes it by interpreting the bytecode. At some point, if a method is run often enough, the dynamic compiler kicks in and converts it to machine code; when compilation completes, it switches from interpretation to direct execution

Cont'd…

- ✓Simple JIT compilers compile all methods before they are executed. This turns out to be wasteful, as many methods are only executed once (or a very few times). In such cases, the time to compile the method can dwarf the time required to execute it. All this compilation also increases memory usage because the compiled code must be stored. As a result, the HotSpot runtime executes many methods in a purely interpreted mode. To ensure maximum performance for these methods, the HotSpot runtime provides a highly optimized bytecode interpreter
- ✓In addition to the bytecode interpreter, the runtime is responsible for memory management and thread synchronization. The HotSpot runtime provides several important optimizations in these areas

Cont’d…

- The term *hot spot* is often used by programmers to describe a piece of code that takes up a large percentage of a program's total execution time. The Java HotSpot virtual machine is Sun Microsystems' advanced JVM implementation. The HotSpot VM provides a dynamic optimizing compiler that looks for hot spots in a program and automatically improves their performance as the program is running-this is where it gets its name

Performance Tuning

HotSpotVM provides a huge list of VM arguments which can be useful for fine tuning and tweaking the VM so as to meet the performance requirements of an application

GC Tuning and Selection

- -XX:+UseSerialGC
  - Default for client class machines
- -XX:+UseParallelGC
  - Default for server class machines
- -XX:UseParallelOldGC
  - Enable Parallel Compacting GC
- -XX:+UseConcMarkSweepGC
  - Enable CMS GC
- -XX:+CMSIncrementalMode
  - Further reduces the pauses by running the concurrent phases incrementally

Cont'd…

- -XX:ParallelGCThreads=n (default is no. of CPU)
- -XX:MaxGCPauseMillis=n
  - Hint to the parallel collector that pause times of n milliseconds or less are desired. The parallel collector will adjust the heap size and other garbage collection-related parameters in an attempt to keep garbage collection pauses shorter than n milliseconds
- -XX:GCTimeRatio=n
  - The ratio of garbage collection time to application time is

    1 / (1 + n)
  - For example -XX:GCTimeRatio=19 sets a goal of 5% of the total time for garbage collection. The default goal is 1% (i.e. **n**= 99). The time spent in garbage collection is the total time for all generations
  - If the throughput goal is not being met, the sizes of the generations are increased in an effort to increase the time the application can run between collections

Cont'd…

- -XX:MinHeapFreeRatio=40 (default) & -XX+MaxHeapFreeRatio=70 (default)
  - Target range for the proportion of free space to total heap size. These are applied per generation. For example, if minimum is 30 and the percent of free space in a generation falls below 30%, the size of the generation is expanded so as to have 30% of the space free. Similarly, if maximum is 60 and the percent of free space exceeds 60%, the size of the generation is shrunk so as to have only 60% of the space free
- -XX:NewSize=n
  - Default initial size of the young generation
- -XX+NewRatio=2 for client VM, 8 for server VM
  - Ratio between the young and old generations. For example, if n is 3, then the ratio is 1:3 and the combined size of Eden and the survivor spaces is one fourth of the total size of the young and old generations

Cont'd…

- -XX:+PrintGCDetails
  - For every collection, this results in the output of information such as the size of live objects before and after garbage collection for the various generations, the total available space for each generation, and the length of time the collection took
- -XX:+PrintGCTimeStamps
  - This outputs a timestamp at the start of each collection, in addition to the information that is output if the command line option –XX:+PrintGCDetails is used
- -XX:-DisableExplicitGC
  - Disables call to System.gc()

Other important options

- Java heap space
  - -Xms=32m –Xmx=128m
    - Setting the minimum and the maximum heap size for the VM
- Perm gen space
  - -XX:MaxPermSize=n
    - This is the area of the heap where the JVM stores it's metadata like the classes and it's reflection
- Stack area
  - -XX:ThreadStackSize=512
- String tuning
  - -XX:+UseCompressedStrings
    - Use byte[] for ASCII Strings
  - -XX:+OptimizeStringConcat

Debugging parameters

- -XX:-PrintCompilation
  - Print message whenever a method is compiled
- -XX:-CITime
  - Prints time spent in JIT compiler
- -XX:ErrorFile=./hs_err_pid<pid>.log
  - If an error occurs, save the error data log to this file
- -XX:HeapDumpPath=./java_pid<pid>.hprof
  - Path to the file or folder for the heap dump
- -XX:-HeapDumpOnOutOfMemoryError
- XX:OnError="<cmd args>;<cmd args>”
  - Execute some user defined command on fatal error
- -XX:OnOutOfMemoryError="<cmd args>; <cmd args>”
- -XX:-PrintClassHistogram
  - Will print histogram of class instances on Ctrl+Break

Some examples

- **java** -Xmx3800m -Xms3800m -Xmn2g -Xss128k
  -XX:+UseParallelGC -XX:ParallelGCThreads=20

- **Xmx3800m -Xms3800m**
  - Configures a large Java heap to take advantage of the large memory system
- **-Xmn2g**
  - Configures a large heap for the young generation (which can be collected in parallel), again taking advantage of the large memory system. It helps prevent short lived objects from being prematurely promoted to the old generation, where garbage collection is more expensive
- **-Xss128k**
  - Reduces the default maximum thread stack size, which allows more of the process' virtual memory address space to be used by the Java heap
- **-XX:+UseParallelGC**
  - Selects the parallel garbage collector for the new generation of the Java heap (note: this is generally the default on server class machines)
- **-XX:ParallelGCThreads=20**
  - Reduces the number of garbage collection threads. The default would be equal to the processor count, which would probably be unnecessarily high on a 32 thread capable system

Cont'd…

- **java** -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC
- **-Xmx3550m -Xms3550m**
  - Sizes have been reduced. The ParallelOldGC collector has additional native, non-Java heap memory requirements and so the Java heap sizes may need to be reduced when running a 32-bit JVM
- **-XX:+UseParallelOldGC**
  - Use the parallel old generation collector. Certain phases of an old generation collection can be performed in parallel, speeding up a old generation collection

Profiling Java Applications

JMX makes it very easily to profile Java applications in the form of MBeans. And with the help of tools like JConsole and VisualVM, we can easily identify bottlenecks in our application

Profiling tools

- Many systems don't meet all of their performance requirements on the first try. Once you've determined that a performance problem exists, you need to begin profiling. Profiling determines what areas of the system are consuming the most resources. Many tools are available to help you with this process. Profilers are most useful for identifying computational performance and RAM footprint issues
- By analyzing data from a profiler, you can isolate the parts of the system that are causing your performance problems. This information can then be used to determine what changes will reap the greatest benefit. Sometimes the solution is as simple as modifying a single method, algorithm, or data structure

Cont'd…

- ✓Measuring method execution times by hand is fine when you suspect a particular method is slow. However, it is more difficult, and usually more important, to find the performance bottlenecks (also known as *hot spots*) in your program. This is where profiling tools come in. There are many profiling tools available-some are included with the Java 2 SDK, and others are stand-alone commercial products

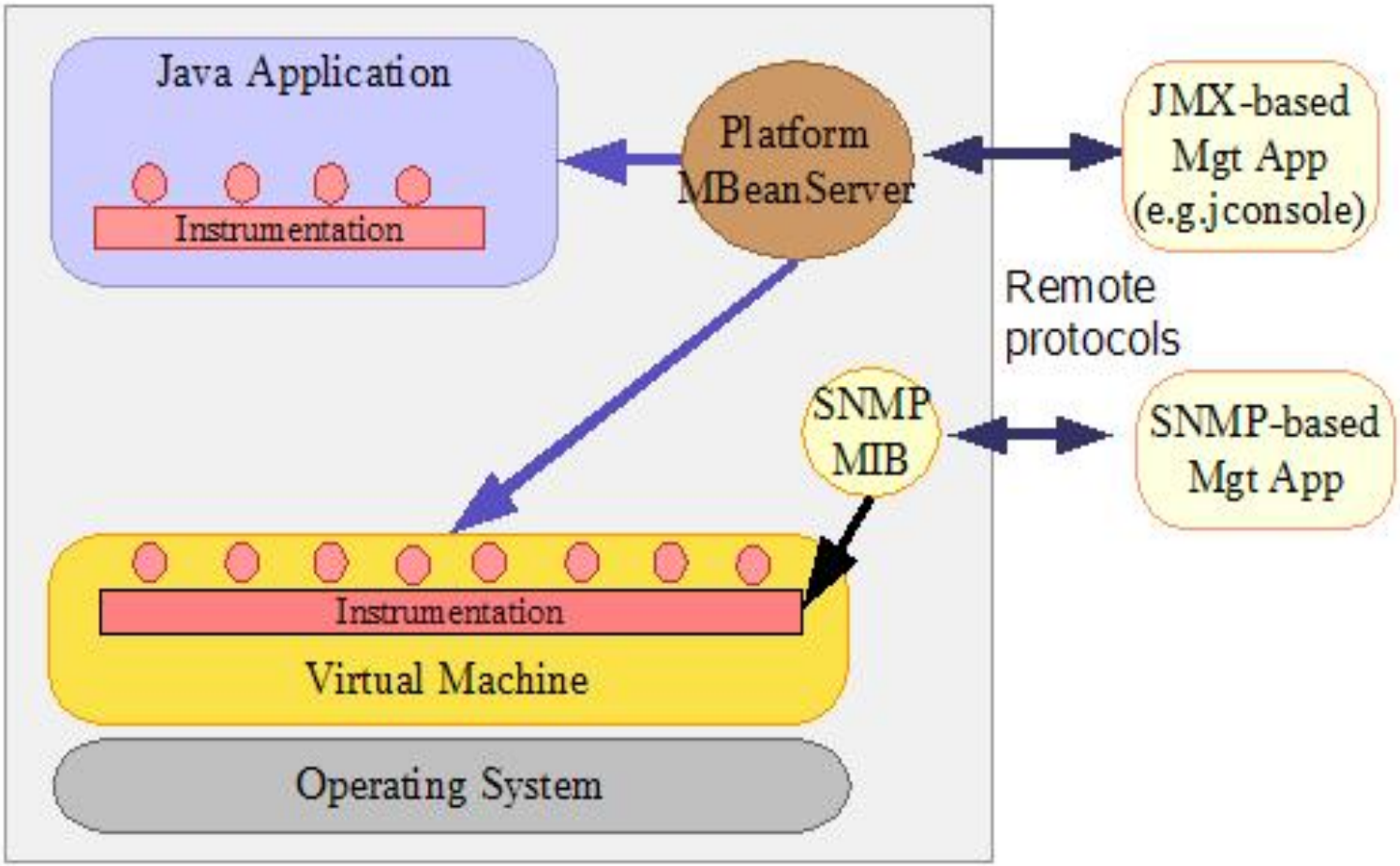
JMX

- ✓ Java Management Extensions provides a standard way of managing resources such as applications, devices, and services
- ✓ Because the JMX technology is dynamic, you can use it to monitor and manage resources as they are created, installed and implemented. You can also use the JMX technology to monitor and manage the Java Virtual Machine (Java VM). Before Java 5, JMX was a part of the J2EE stack and was primarily used for monitoring and managing servers and components

Cont'd…

- Using the JMX technology, a given resource is instrumented by one or more Java objects known as *Managed Beans*, or *MBeans*. These MBeans are registered in a core-managed object server, known as an *MBean server*. The MBean server acts as a management agent and can run on most devices that have been enabled for the Java programming language

JMX and Java 5



java.lang.instrument

Interface Summary	
ClassLoadingMXBean	The management interface for the class loading system of the Java virtual machine.
CompilationMXBean	The management interface for the compilation system of the Java virtual machine.
GarbageCollectorMXBean	The management interface for the garbage collection of the Java virtual machine.
MemoryManagerMXBean	The management interface for a memory manager.
MemoryMXBean	The management interface for the memory system of the Java virtual machine.
MemoryPoolMXBean	The management interface for a memory pool.
OperatingSystemMXBean	The management interface for the operating system on which the Java virtual machine is running.
RuntimeMXBean	The management interface for the runtime system of the Java virtual machine.
ThreadMXBean	The management interface for the thread system of the Java virtual machine.

With the help of these powerful MBeans available, we can easily collect all the vital statistics for which we had to knock the JNI door and rely on third party tools in the past

Cont'd…

- ✓jConsole was introduced to help Java developers profile their applications
- ✓VisualVM introduced in Java 6 is a much better alternative to jConsole

Profiling Java applications

Lab session

Conclusion

See you next time!