Lecture 5

Collections, Maps and Threads
What are collections?

• a type of class that stores and manipulates groups of objects.
• Java 1.2 introduced the Collections framework
• maps, dictionaries and associative arrays
• collections are containers of objects a.k.a elements
Collections

• extend the Collection interface
• Collections are dynamic in size (unlike arrays).
• List – dups ok
  – ArrayList – dynamic length
  – LinkedList – order preserved
• Set – no dups
  – HashSet - efficient storage of an element
  – LinkedHashSet – insertion order is preserved
different types of collections/maps

- **Array** – fixed size stores dups and nulls (not really a collection)
- **List** – variable size, specific insertion order, duplicates allowed (abstract)
- **Set** - variable size, specific insertion order, NO duplicates allowed (abstract)
- **ArrayList** – dynamic length array.
- **LinkedList** – a list with a preserved order. Supports queues and stacks.
- **Vector** – an array that grows and shrinks. Synchronized access.
- **HashSet** – stored via a hashing mechanism. Fast access. Order is not guaranteed over time through iterator. Membership in set is quickly determined.
- **TreeSet** – elements are stored in sorted order using a balance tree algorithm. Order is ascending.
- **HashMap** – (key, object reference) pairs stored. Retrieved by key. Order not preserved. Access to elements optimized by hashing the key and placing elements into buckets
- **LinkedHashSet** – a set that is managed as a hash with a doubly linked list. Order is maintained. Order is determined by insertion.
Collection interface

• Collection
  - List is a Collection
  - Set is a Collection
  - SortedSet is a Collection

• AbstractList implements Collection, List
  - ArrayList is a AbstractList
  - Vector is a AbstractList
methods in the interface

boolean add(Object o)                //Ensures that this collection contains the specified
   // element (optional operation).
boolean addAll(Collection c)         //Adds all of the elements in the specified collection to
   // this collection (optional).
void clear()                        //Removes all of the elements (optional)
boolean contains(Object o)          //true if this collection contains the specified element.
boolean containsAll(Collection c)    //true if this collection contains all of the elements
   //in the specified collection.
boolean equals(Object o)            //Compares the specified object with this collection for
   //equality.
int hashCode()                      //Returns the hash code value for this collection.

*optional methods not supported must throw the UnsupportedOperation
   Exception in their implementation
methods in the interface(2)

boolean isEmpty() //Returns true if this collection contains no elements.

Iterator iterator() //Returns an iterator over the elements in this collection.

boolean remove(Object o) //Removes a single instance of the specified element
// from this collection, if it is present (optional)

boolean removeAll(Collection c) // Removes all this collection's elements that are
// also contained in the specified collection (optional)

boolean retainAll(Collection c) //Retains only the elements in this collection that are
// contained in the specified collection (optional)

int size() //Returns the number of elements in this collection.

Object[] toArray() //Returns an array containing all of the elements

Object[] toArray(Object[] a) //Returns an array containing all of the elements in this
//collection; the runtime type of the returned array is
//that of the specified array.
//AUTOMATICALLY reallocates the array if the supplied
//one is too small!
Iterator interface

- Supports getting the objects in a collection
- java.util package

boolean hasNext() //Returns true if the iteration has more elements.
Object next() //Returns the next element in the iteration.
void remove() //Removes from the underlying collection the
//last element returned by the iterator (optional)

throws
UnsupportedOperationException
IllegalStateException
ListIterator interface

- java.util package
- public interface ListIterator
  - extends Iterator

  void add(Object o) // Inserts the specified element into the
                    // list (optional operation).

  boolean hasNext() // Returns true if this list iterator has
                    // more elements when traversing the list
                    // in the forward direction.

  boolean hasPrevious() // Returns true if this list iterator has
                        // more elements when traversing the list
                        // in the reverse direction.

  Object next() // Returns the next element in the list.

  int nextIndex() // Returns the index of the element that
                  // would be returned by a subsequent call
                  // to next.

  Object previous() // The previous element in the list.
ListIterator interface (2)

int previousIndex()   // Returns the index of the element that
                      // would be returned by a subsequent call
                      // to previous.

void remove()         // Removes from the list the last element
                      // that was returned by next or previous
                      // (optional)

void set(Object o)    // Replaces the last element returned by
                      // next or previous with the specified
                      // element (optional operation).

throws
UnsupportedOperationException
IllegalStateException
NoSuchElementException
ClassCastException
IllegalArgumentException
CollectionsDemo

- ArrayList myListOfNames;
- HashSet myHashSetOfNames;
- HashMap myHashMapOfStudentInfo;
- Hashtable myHashtableOfStudentInfo;
- TreeSet myTreeSetOfNames;
CollectionsDemo

********************************
ListArray Demo
The iterator.next() returned: Smith
The iterator.next() returned: Chang
The iterator.next() returned: Chen
The iterator.next() returned: Lewin
The iterator.next() returned: Peng
The iterator.next() returned: Chen
====
The iterator.previous() returned: Chen
The iterator.previous() returned: Peng
The iterator.previous() returned: Lewin
The iterator.previous() returned: Chen
The iterator.previous() returned: Chang
The iterator.previous() returned: Smith
CollectionsDemo

*******************************
HashSet Demo
add() failed on Chen
The iterator.next() returned: Peng
The iterator.next() returned: Chen
The iterator.next() returned: Chang
The iterator.next() returned: Lewin
The iterator.next() returned: Smith
====
myHashSetOfNames.size() = 5
The iterator.next() returned: Peng
The iterator.next() returned: Chen
  remove the current element
myHashSetOfNames.size() = 4
The iterator.next() returned: Peng
The iterator.next() returned: Chang
The iterator.next() returned: Lewin
The iterator.next() returned: Smith
CollectionsDemo

*************************
HashMap Demo
The put (Chang , Chang 212-000-0000 0)
The put (Khan , Khan 212-111-1111 1111)
The put (Donovan , Donovan 212-222-2222 2222)
The put (Patel , Patel 212-333-3333 3333)
The put (Poelman , Poelman 212-444-4444 4444)
The put (Aja , Aja 212-555-5555 5555)
The put (Wu , Wu 212-666-6666 6666)
The put (Josie , Josie 212-777-7777 7777)
The put (Fez , Fez 212-999-9999 9999)
====
Find entries in the map
The myHashMapOfStudentInfo.get (Chang) = Chang 212-000-0000 0
The myHashMapOfStudentInfo.get (Khan) = Khan 212-111-1111 1111
The myHashMapOfStudentInfo.get (Donovan) = Donovan 212-222-2222 2222
The myHashMapOfStudentInfo.get (Patel) = Patel 212-333-3333 3333
The myHashMapOfStudentInfo.get (Poelman) = Poelman 212-444-4444 4444
The myHashMapOfStudentInfo.get (Aja) = Aja 212-555-5555 5555
The myHashMapOfStudentInfo.get (Wu) = Wu 212-666-6666 6666
The myHashMapOfStudentInfo.get (Josie) = Josie 212-777-7777 7777
The myHashMapOfStudentInfo.get (Fez) = Fez 212-999-9999 9999
Search for Chang
key = Chang => Chang 212-000-0000 0
Search for Chang
no object returned
====
CollectionsDemo

*************************
Hashtable Demo
The put (Chang, Chang 212-000-0000 0)
The put (Khan, Khan 212-111-1111 1111)
The put (Donovan, Donovan 212-222-2222 2222)
The put (Patel, Patel 212-333-3333 3333)
The put (Poelman, Poelman 212-444-4444 4444)
The put (Aja, Aja 212-555-5555 5555)
The put (Wu, Wu 212-666-6666 6666)
The put (Josie, Josie 212-777-7777 7777)
The put (Fez, Fez 212-999-9999 9999)
====
CollectionsDemo

Find entries in the table

Tjava.lang.ClassCastException: Lecture5.CollectionsDemo$StudentInfo
    at java.util.TreeMap.compare(TreeMap.java:1047)
    at java.util.TreeMap.put(TreeMap.java:449)
    at java.util.TreeSet.add(TreeSet.java:198)
    at Lecture5.CollectionsDemo.demoTreeSetClass(CollectionsDemo.java:282)
    at Lecture5.CollectionsDemo.main(CollectionsDemo.java:359)

Exception in thread "main"

The myHashtableOfStudentInfo.get(Chang) = Chang 212-000-0000 0
The myHashtableOfStudentInfo.get(Khan) = Khan 212-111-1111 1111
The myHashtableOfStudentInfo.get(Donovan) = Donovan 212-222-2222 2222
The myHashtableOfStudentInfo.get(Patel) = Patel 212-333-3333 3333
The myHashtableOfStudentInfo.get(Poelman) = Poelman 212-444-4444 4444
The myHashtableOfStudentInfo.get(Aja) = Aja 212-555-5555 5555
The myHashtableOfStudentInfo.get(Wu) = Wu 212-666-6666 6666
The myHashtableOfStudentInfo.get(Josie) = Josie 212-777-7777 7777
The myHashtableOfStudentInfo.get(Fez) = Fez 212-999-9999 9999
Search for  Chang
key = Chang => Chang 212-000-0000 0
Search for  ChangX
no object returned

====
Map Interface

- not a collection
- public class HashMap
  - extends AbstractMap
  - implements Map, Cloneable, Serializable

- key and value pairs stored in the map (dictionary)
- Map implemented in these classes
  - AbstractMap
    - HashMap
    - IdentityHashMap
    - TreeMap
    - WeakHashMap
  - Hashtable a subclass of Dictionary
methods

clear() //Removes all mappings from this map (opt)
boolean containsKey(Object key) //true if this map contains a mapping for
  // the specified key.
boolean containsValue(Object value) //true if this map maps one or more
  //keys to the specified value.
Set entrySet() //a set view of the mappings contained in
  //this map.
boolean equals(Object o) //Compares the specified object with this
  //map for equality.
Object get(Object key) //Returns the value to which this map maps
  //the specified key.
int hashCode() //Returns the hash code value for this map.
boolean isEmpty() //true if this map contains no key-value
  //mappings.
Set keySet() //Returns a set view of the keys contained
  //in this map.
Object put(Object key, Object value) //Associates the specified value
  //with the specified key in this map (opt)
  //(optional operation).
methods(2)

void putAll(Map t) //Copies all of the mappings from the
//specified map to this map (opt)

Object remove(Object key) //Removes the mapping for this key from this
//map if it is present (opt).

int size() //Returns the number of key-value mappings in
//this map.

Collection values() //Returns a collection view of the values
//contained in this map.
HashMap Class

- **Load Factor & Initial Size**
  - number of entries in the hashtable exceeds the product of the load factor and the current capacity
  - Will rehash if load factor exceeds specified
  - Takes time
Hashtable Class

• Like a HashMap but older version
• Synchronized
  – Thread safe access
• No nulls
• Load Factor & Initial Size
  – number of entries in the hashtable exceeds the product of the load factor and the current capacity
  – Will rehash if load factor exceeds specified
  – Takes time
Hashtable Class

Hashtable() //Constructs a new, empty hashtable with a default //initial capacity (11) and load factor, which is //0.75.

Hashtable(int initialCapacity) //Constructs a new, empty //hashtable with the specified initial capacity and //default load factor, which is 0.75.

Hashtable(int initialCapacity, float loadFactor) //Constructs a //new, empty hashtable with the specified initial //capacity and the specified load factor.

Hashtable(Map t) //Constructs a new hashtable with the same //mappings as the given Map.
Threads

- A unit of execution within a process
- Process may have multiple threads
- Process holds heap, file handles tables etc. Shared across all threads in process.
- Non-atomic operations executed on different threads at the same time can result in unpredictable outcomes
  - Atomic operations happen in a single operation
  - \( x = y; \) //atomic if \( x\)\&\( y \) are ints NOT if longs or doubles!
  - Why? Int is 4 bytes long is 8. Most processors work on 4 bytes per instruction cycle NOT 8 or more.
Threads

• Threads have their own:
  – Instruction Counter
  – Stack
  – Local Variables

• Threads share:
  – Process variables
  – Code
  – Heap - Class and Instance Variables
Threads vs. Processes

• Quick communication between threads because they share memory space
• Lighter weight
• Threads share code, objects in memory, processes don’t
Threads in Java

• Platform dependent!
  – some are non-preemptive and some preemptive
• Swing uses a single thread
  – different from the main thread
• Threads share heap objects
  – created via new
• Threads have their own stack
• Threads have private instances of stack variables and local variable in a method
Scheduling

• Preemptive
  – OS switches between threads running based on timeslice and priority
  – Supports concurrent and parallel execution
  – More flexible
  – Easier for developer to implement

• Cooperative
  – threads yield control and application switches to executing a different thread
  – Supports concurrent only – no multiple CPU support
  – Faster
  – Requires coding yield()s into the application code
Relationship between Process and its threads
Multiple processes share nothing but the JVM and OS.
Thread safe

- synchronized
- atomic variables
- immutable objects
- mutexes
thread priority

• higher priority gets more time on the CPU
• Java has 10 levels: 0 - 9
  – Solaris $2^{31}$ & processes have priorities
  – NT 7 levels plus boosting w/no process priorities!
  – Using priority under Java is not reliable across platforms
thread priority (2)

- Thread.MAX_PRIORITY
  - //The maximum priority that a thread can have.
- Thread.MIN_PRIORITY
  - //The minimum priority that a thread can have.
- Thread.NORM_PRIORITY
  - //The default priority that is assigned to a thread.
Using priorities

• Highest for UI
• Medium for calculations
• Low background tasks and clean up
Daemon threads

• Continue to live after the process that created them terminates (unlike regular threads)
• Background tasks and services
• When all processes in the JVM terminate the daemon threads will be stopped by JVM
• use setDaemon(true) before you start() the thread
join()

• make one thread wait to finish until the other does
• use instead of polling to see if all the threads finished
Creating threads

• **FourThreadDemo sample**
• Creates four objects and runs them on four separate threads
• Output:

12341234123412341234123412341234123412341234

-------------

Threads should stop now please

-------------
FourThreadDemo

• “12341234...” isn’t always the order to expect, especially on a multiprocessor machine

• Sleep time is quite long compared to execution. Shorter sleep will result in less predictable output.

• Separate objects on separate threads will only have synchronization issues on the class variables and methods!
FourThreadDemo_SameObject

- Single object executing on four threads concurrently
- Shows thread shared and private variables
- *thread “label” [counter] thread name*
- Each thread increments the same counter variable

1[0] - Limberger
1[1] - Gruyere
1[2] - Gorgonzola
1[3] - Fontina
1[4] - Limberger
1[5] - Gruyere
1[6] - Gorgonzola
1[7] - Fontina
1[8] - Limberger
1[9] - Gruyere
FourThreadDemo_SameObject

- If you set the sleep time to 5 milliseconds instead of 500 things are different!
- Remember incrementing a long isn’t an atomic operation

...  
1[9] - Gruyere  
1[10] - Gorgonzola  
1[10] - Fontina  
1[10] - Limberger  
1[10] - Gruyere  
1[14] - Gorgonzola  
1[15] - Limberger  
1[16] - Gruyere  
1[17] - Fontina  
1[18] - Gorgonzola  
1[18] - Limberger  
1[18] - Gruyere  
1[18] - Fontina  
1[22] - Limberger  
1[23] - Gorgonzola
Execution of threads

- 1 CPU – interleaved (timesliced)
- $n$ CPUs – in parallel (depends on the JVM)
- Threads can be executed in any sequence – not just sequentially (round robin)
- One thread can change part of an object while a different part is being changed by a different thread unless synchronized
- Bugs can be hard to reproduce and fix
ThreadContentionDemo

• Output:

No thread contention

-----------
Pass 0: Time lost (non vs. sync-ed): 30 ms. 175% increase
Pass 1: Time lost (non vs. sync-ed): 1 ms. 101.43% increase
Pass 2: Time lost (non vs. sync-ed): 40 ms. 233.33% increase
Pass 3: Time lost (non vs. sync-ed): 0 ms. 100% increase
Pass 4: Time lost (non vs. sync-ed): 40 ms. 233.33% increase
Pass 5: Time lost (non vs. sync-ed): 0 ms. 100% increase
Pass 6: Time lost (non vs. sync-ed): 41 ms. 236.67% increase

With thread contention

-----------
Pass 7: Time lost (non vs. sync-ed): 711 ms. 1,522% increase
Pass 8: Time lost (non vs. sync-ed): 791 ms. 2,736.67% increase
Ways to use threads

• Extend the Thread class
• Implement the Runnable interface
Thread class

- public class Thread
  - extends Object
  - implements Runnable

- Thread()
  - Allocates a new Thread object.

- Thread(Runnable target)
  - Allocates a new Thread object.

- Thread(Runnable target, String name)
  - Allocates a new Thread object.

- Thread(String name)
  - Allocates a new Thread object. Give a name to the thread.
Thread class (2)

- static int activeCount() //number of active threads
- static Thread currentThread()
- void destroy() //Destroys this thread, without any cleanup.
- String getName()
- int getPriority()
- ThreadGroup getThreadGroup() //thread group to which this thread belongs.
- boolean isAlive() //Tests if this thread is alive.
- boolean isDaemon()
- void join() //Waits for this thread to die.
- void run() //If this thread was constructed using a separate Runnable run object, then that Runnable object's run method // is called; otherwise, this method does nothing and returns.
- void setDaemon(boolean on)
- void setName(String name)
- void setPriority(int newPriority)
Thread class (3)

- static void sleep(long millis)
- void start() //Causes this thread to begin execution; the Java //Virtual Machine calls the run method of this thread.
- void stop() //Deprecated. This method is inherently unsafe.
- void suspend() //Deprecated. This method has been deprecated, //as it is inherently deadlock-prone.
- static void yield() //Causes the currently executing thread //object to temporarily pause and allow other threads to //execute.

- Rarely do you extend Thread
- So how do you stop a thread?
Thread class methods

- `currentThread();`
- `yield(); // need this for some platforms (Solaris)`
- `sleep(); // number of milliseconds and implicitly yields`
- `or blocking IO calls`
Implementing the **Runnable** interface

class MyNewClass implements Runnable
{
    void run()
    {
    }
}

public static void main(String[] args)
{
    MyNewClass instanceOfMyNewClass = new MyNewClass();
    Thread newThread = new Thread(instanceOfMyNewClass);
    newThread.start();
}

Swing

- uses a single thread \texttt{AWT-EventQueue-0}
- automatically creates that thread
- \textbf{all} parts of your swing GUI run on that thread
- Swing is not synchronized
Hang demo

• Sleep button blocks the swing thread for 5 seconds, queuing up button presses.

• Output:

Hello world
Hello world
Hello world
ThreadDemo2

- Shows shared variables
- Main thread updates the counter field once per second
- The AutoIncrementer object will update it. Shows single Swing thread and spawning separate AutoIncrementer object and thread. It increments and updates the counter every 1/10 second.
- AutoIncrementer is related to different JFrames
- Even when you create a new thread with a dialog it will be on the same single swing thread.
Why use threads at all?

- Multiple CPUs can do work in parallel
- Improve responsiveness of an application
- Efficiently utilize CPU with tasks that block for periods of time (waiting for a key press, for example).
- Better design
- Support multiple users more easily
Why use threads at all?(2)

• Garbage collection
• Background processes – rebalancing a tree, rehashing a HashMap
• Support polling with stopping your whole program
• Animations
• Simulations
synchronized key word

- methods
- code blocks
- objects
- a mutex is required to gain access to the code block. Only one thread can hold the mutex at a time.
- synchronization can be 150 times slower than non-synchronized access!
- make sync blocks as small as possible
**synchronized keyword (2)**

- not part of a method signature and thus is NOT inherited!
- Mutex is released on a return, break or exception
- You need do nothing but the `synchronized`
- Less flexible but easier to use
- Locks the object instance NOT the code! – different instance with a synchronized method will execute in parallel with no blocking from the synchronized.
synchronized keyword (3)

• All the synchronized instance methods and instance variables on an object share a single mutex
• All the synchronized class methods and class variables on an class share a single mutex (a different one from each object).
• So, when a thread acquires that mutex all other threads are blocked from executing and accessing the other methods and variables!
Volatile keyword

• tells java not to optimize this chunk of code
• Even though access to a variable is atomic optimizations might break your program logic.
• `boolean myBoolean; //could get loaded in a register. Registers aren’t shared across threads. Could lead to bugs. Especially in a multiple CPU environment (actually volatile doesn’t quite work right for multiple CPUs).
• `volatile boolean myBoolean;`
ThreadContentionDemo

- Shows the cost of threads competing for a mutex on execution time of non-synchronized vs. synchronized methods
- Output:

  No thread contention
  -----------
  Pass 0: Time lost (non vs. sync-ed): 30 ms. 175% increase
  Pass 1: Time lost (non vs. sync-ed): 30 ms. 175% increase
  Pass 2: Time lost (non vs. sync-ed): 20 ms. 150% increase
  Pass 3: Time lost (non vs. sync-ed): 40 ms. 233.33% increase
  Pass 4: Time lost (non vs. sync-ed): 30 ms. 175% increase
  Pass 5: Time lost (non vs. sync-ed): 51 ms. 227.5% increase
  Pass 6: Time lost (non vs. sync-ed): 20 ms. 150% increase
  With thread contention
  -----------
  Pass 7: Time lost (non vs. sync-ed): 1,072 ms. 1,074.55% increase
  Pass 8: Time lost (non vs. sync-ed): 1,152 ms. 1,745.71% increase
is the following class thread safe?

class X
{
    private static int Age;
    public synchronized int incrementAge()
    {
        age++;
        return age;
    }
}
Deadlocks

synchronized public methodA()
{
    methodB();
}

synchronized public methodB()
{
    methodA();
}

threadA.methodA();                threadB.methodB();
When can I ignore thread issues?

- single threaded applications (duh!)
- reads of immutable objects
- constructors (as long as they don’t touch class variables)
- Objects only used by a single thread
- create immutable objects – all fields `final`
- use a synchronization wrapper when needed
  - see the collection classes
  - the decorator pattern from GOF
two phase locking

• if locking multiple objects first acquire ALL the locks
• then do your operations on the objects
• then release the locks in the opposite order of acquiring them
• Always acquire them in the same order through out your program
• Never release some locks and acquire new ones, if you can avoid it.
UsingTheThreadClassDemo

- Shows the Thread class
- Shows signaling a thread by `interrupt()` and checking a boolean variable in the `catch()` to see if the thread should terminate
in the `main()`

myUsingTheThreadClassDemo.keepRunning=false;
myUsingTheThreadClassDemo.interrupt();

• This code throws an exception via the `interrupt()` that gets `catch()`-ed by the thread object.
infinite loop in the thread object

while(true)
{
    try{
        System.out.println("THREAD - doing something and about to sleep");
        Thread.currentThread().sleep(5000); //every 5 second wake up
        System.out.println("THREAD - done sleep() ing");
    }catch(InterruptedException e){
        System.out.println("THREAD - INTERRUPTED : "+e);
        System.out.println("THREAD - threadName = " +
                          Thread.currentThread().getName());
        if (!keepRunning)
        {
            System.out.println("THREAD - terminating the thread");
            break; //jumps out of the while loop and we gracefully end
        }
    }
}
communicating with a thread

- `wait();`
- `notify(), notifyAll();`
Summary

- Different types of collections are available through the JDK
  - Arrays, Lists, Sets and Maps
  - ArrayList, LinkedList, Vector, HashSet, TreeSet, HashMap, LinkedHashSet, Hashtable
Summary

• Threads
  – Not standard on the platforms
  – concurrency and parallelism

• Scheduling
  – Preemptive vs. Cooperative

• Swing runs on a single and different thread

• synchronized & mutexes
  – Class has a shared mutex
  – Each object instance has a different mutex

• Deadlocks still possible

• Two phase locking
Resources

• Design patterns
  – GOF http://hillside.net/patterns/DPBook/DPBook.html