Programming Languages and Techniques (CIS120)

Lecture 37
December 3, 2018
Advanced Java Miscellany (Hashing)
Announcements

HW09: Game of your own
   – Due Monday, December 10\textsuperscript{th} at 11:59pm
   – No late submissions!

Final Exam:
   • Thursday, December 20\textsuperscript{th} 6:00-8:00 PM
   • 6 Locations (!):
     – Towne 100  Last Names A – F
     – DRLB A1    Last Names G – L
     – DRLB A2    Last Names M – O
     – DRLB A4    Last Names P – R
     – DRLB A6    Last Names S
     – DRLB A8    Last Names T – Z
   • Makeup exam – form available on the course web site

   • Mock Exam and Review Session:
     Wednesday, December 12\textsuperscript{th} 1:00-4:00PM, Towne 100
Advanced Java Miscellany

- Hashing: HashSets & HashMaps
- Threads & Synchronization
- Garbage Collection
- Java 1.8 Lambdas
- Packages
- JVM (Java Virtual Machine) and compiler details:
  - class loaders, security managers, just-in-time compilation
- Advanced Generics
  - Bounded Polymorphism: type parameters with ‘extends’ constraints
    class C<A extends Runnable> { ... }
  - Type Erasure
  - Interaction between generics and arrays
- Reflection
  - The Class class

We’ll touch on these.

For all the nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs
Hash Sets & Hash Maps

array-based implementation of sets and maps
Hash Sets and Maps: The Big Idea

Combine:

• the advantage of arrays:
  – *efficient* random access to its elements

• with the advantage of a map datastructure
  – arbitrary keys (not just integer indices)

How?

• Create an index into an array by *hashing* the data in the key to turn it into an int
  – Java’s hashCode method maps key data to ints
  – Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique
A schematic HashMap taking Strings (student names) to Undergraduate Majors. The hashCode takes each string name to an integer code, which we then take “mod 256” to get an array index between 0 and 255.
For example, “John Doe”.hashCode() mod 256 is 254.
Hash Collisions

• Uh Oh: Indices derived via hashing may not be unique!
  
  “Jane Smith”.hashCode() % 256 \rightarrow 253
  “Joe Schmoe”.hashCode() % 256 \rightarrow 253

• Good hashCode functions make it \textit{unlikely} that two keys will produce the same hash

• But, it can still sometimes happen that two keys produce the same index – that is, their hashes \textit{collide}
Here, “Jane Smith”.hashCode() and “Joe Schmoe”.hashCode() happen to collide. The bucket at the corresponding index of the Hash Map array stores the map data.
Bucketing and Collisions

• Using an array of buckets
  – Each bucket stores the mappings for keys that have the same hash.
  – Each bucket is itself a map from keys to values (implemented by a linked list or binary search tree).
  – The buckets can’t use hashing to index the values – instead they use key equality (via the key’s equals method)

• To look up a key in the Hash Map:
  1. Find the right bucket by indexing the array through the key’s hash
  2. Search linearly through the bucket contents to find the value associated with the key

• Not the only solution to the collision problem
What gets printed to the console?

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere else...
Map<Point,String> m = new HashMap<Point,String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));
```

True
False
I have no idea
4
Hashing and User-defined Classes

public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }

    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere else...
Map<Point, String> m = new HashMap<Point, String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));

What gets printed to the console?
1. true
2. false
3. I have no idea

ANSWER: 2 – hashCode not implemented
hashCode Requirements

Whenever you override `equals` you must also override `hashCode` in a consistent way:
- whenever `o1.equals(o2) == true` you must ensure that `o1.hashCode() == o2.hashCode()`

Why? Because comparing hashes is supposed to be a quick approximation for equality.

• Note: the converse does not have to hold:
  - `o1.hashcode() == o2.hashCode()` does not necessarily mean that `o1.equals(o2)`
Example for Point

```java
public class Point {
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
}
```

• Examples:
  – (new Point(1,2)).hashCode() yields 994
  – (new Point(2,1)).hashCode() yields 1024

• Note that equal points (in the sense of `equals`) have the same `hashCode`

• Why 31? Prime chosen to create more uniform distribution

• Note: Tools (e.g. eclipse) can generate this code
Recipe: Computing Hashes

• What is a good recipe for computing hash values for your own classes?
  – intuition: “smear” the data throughout all the bits of the resulting

1. Start with some constant, arbitrary, non-zero int in \texttt{result}.
2. For each significant field \( f \) of the class (i.e. each field taken into account when computing equals), compute a “sub” hash code \( c \) for the field:
   – For boolean fields: \((f \ ? \ 1 \ : \ 0)\)
   – For byte, char, int, short: \((\text{int}) \ f\)
   – For long: \((\text{int}) (f \ ^ \ (f \ >>> \ 32))\)
   – For references: \(0\) if the reference is null, otherwise use the \texttt{hashCode()} of the field.
3. Accumulate those subhashes into the result by doing (for each field’s \( c \)):
   \[
   \text{result} = \text{prime} \ast \text{result} + c;
   \]
4. return \texttt{result}
Hash Map Performance

- Hash Maps can be used to efficiently implement Maps and Sets
  - There are many different strategies for dealing with hash collisions with various time/space tradeoffs
  - Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)
  - See CIS 121 for more info!

- If the hashCode function gives a good (close to uniform) distribution of hashes, the buckets are expected to be small (only one or two elements)
- If the hashCode function gives a bad distribution (e.g. return 0;), the buckets will be large (and performance will be bad)

- Performance depends on workload
NOTE: Terminological Clash

• The word "hash" is also used in cryptography
  – SHA-1, SHA-2, SHA-3, MD5, etc.

• All hash functions reduce large objects to short summaries

• Cryptographic hashes have some extra requirements:
  – Are "one way" (i.e. very hard to invert)
  – Should only very rarely have collisions
  – Are considerably more expensive to compute than hashCode (so not suitable for hash tables)

• Never use hashCode when you need a cryptographic hash!
  – See CIS 331 for more details
Hashing: take away lessons

equals
hashCode
compareTo
Collections Requirements

• All collections invoke equals method on elements
  – Defaults to == (reference equality)
  – Override equals to create structural equality
  – Should always be an equivalence relation: reflexive, symmetric, transitive

• HashSets/HashMaps also invoke hashCode method on elements
  – Override when equals is overridden
  – Should be “compatible” with equals
  – Should try to distribute hash codes uniformly
  – Iterators are not guaranteed to follow order of hashCodes

• Ordered collections (TreeSet, TreeMap) require element type to implement Comparable interface
  – Provide compareTo method
  – Should implement a total order
  – Should be compatible with equals
    • (i.e. o1.equals(o2) exactly when o1.compareTo(o2) == 0)
Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)
Threads

• Java programs can be *multithreaded*
  – more than one “thread” of control operating simultaneously

• A `Thread` object can be created from any class that implements the `Runnable` interface
  – `start`: launch the thread
  – `join`: wait for the thread to finish

• Abstract Stack Machine:
  – Each thread has its own workspace and stack
  – All threads *share* a common heap
  – Threads can communicate via shared references
Uses + Perils

• Threads are useful when one program needs to do multiple things simultaneously:
  – game animation + user input
  – chat server interacting with multiple chat clients
  – hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)

• Problem: Race Conditions
  – What happens when one thread tries to read a memory location at the same time another thread is writing it?
  – What if more than one thread tries to write different values at the same time?
interface Counter {
    public void inc();
    public int get();
}

class UCounter implements Counter {
    private int cnt = 0;

    public void inc() {
        cnt = cnt + 1;
    }

    public int get() {
        return cnt;
    }
}
// The computation thread simply increments
// the provided counter 1000 times

class CounterUser implements Runnable {
    private Counter c;
    private int id;

    CounterUser(int id, Counter c) {
        this.id = id;
        this.c = c;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            // System.out.println("Thread: " + id);
            c.inc();
        }
    }
}
First Try: Two Threads & One Counter

```java
public class MultiThreaded {

    public static void main(String[] args) {
        Counter c = new UCounter();

        // set up a race on the shared counter c
        Thread t1 = new Thread(new CounterUser(1, c));
        Thread t2 = new Thread(new CounterUser(2, c));
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
        }
        System.out.println("Counter value = "+c.get());
    }
}
```
What behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ???" for some other number ???
4. The program will throw an exception.
Answer: The program with print “Counter value = val” for 1000 <= val <= 2000. The answer will likely be different each time the program is run!!!!
Both threads invoke the inc method of a shared counter object. The individual instructions of this method interleave such that they both read 0 and write 1.
The synchronized keyword

- Synchronized methods are *atomic*
  - They run without any other threads running

- Careful use will eliminate races

- Tradeoff:
  - less concurrency means worse performance
/This class uses synchronization

class SynchronizedCounter implements Counter {
    private int cnt = 0;

    public synchronized void inc() {
        cnt = cnt + 1;
    }

    public synchronized int get() {
        return cnt;
    }
}
public class MultiThreaded {

    public static void main(String[] args) {

        Counter c = new SynchronizedCounter();

        // set up a race on the shared counter c
        Thread t1 = new Thread(new CounterUser(1, c));
        Thread t2 = new Thread(new CounterUser(2, c));
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
        }

        System.out.println("Counter value = " + c.get());
    }
}

CIS120
Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"

2. The program will print "Counter value = 2000"

3. The program will print "Counter value = ?????" for some other number ????

4. The program will throw an exception.

Answer: The program will print “Counter value = 2000” every time.
Other Synchronization in Java

Need *thread safe* libraries:
- java.util.concurrent has BlockingQueue and ConcurrentHashMap
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

- Java also provides *locks*
  - objects that act as synchronizers for blocks of code

- **Deadlock**: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A
Immutability!

• Note that read-only datastructures are immune to race conditions
  – It’s OK for multiple threads to read a heap location simultaneously
  – Less need for locking, synchronization

• As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library
• Library written in Haskell
• Concurrency / Distributed Database
• https://github.com/facebook/Haxl