Principles of Software Construction: Objects, Design, and Concurrency

Part 3: Design case studies

Performance

Charlie Garrod **Michael Hilton**

School of Computer Science

institute for SOFTWARE RESEARCH

Administriva

- Homework 4b due Thursday, October 19th
- Reading due today: Effective Java, Items 5, 6, and 51

Feedback from early informal feedaback

- Slides:
  - Post sooner after class
  - Handwritten notes scanned at the end of slides
- Homework clarity concerns
  - Attempting to balance need for clarity with learning skill of gleaning requirements from real problem descriptions
  - Learning goals are explicit, and the basis for evaluation

Early Informal Feedback Continued

- Random Calling:
  - Supports our goal of everyone participating in class
    - We want to hear from everyone, not just the most vocal students
    - Wrong Answers & Mistakes are Expected and Valued
    - Goal is learning, not evaluation
  - Feedback: slows down the class
    - Make ”pass” a more common and expected answer
    - Other suggestions?

Class Parti
and /

My perception is th:
manding student parti
suggested to me that n

Framing Classroom Climate for Student Learning and Retention in Computer Science

Lecia J. Barker
University of Texas
1616 Guadalupe Street
Austin, TX 78701-1213
+1-512-232-8364
lecia@ischool.utexas.edu

Melissa O’Neill
Harvey Mudd College
1250 North Dartmouth Avenue
Claremont, CA 91711-5980
+1-909-607-9661
oneill@cs.hmc.edu

Nida Kazim
University of Texas
1616 Guadalupe Street
Austin, TX 78701-1213
+1-512-471-3821
nida.kazim@gmail.com

ABSTRACT
Despite the best laid plans, counterproductive student behavior can interfere with faculty establishment of supportive classroom climates. This paper describes methods for framing the climate of the computer science classroom to minimize outspoken students’ unwanted displays of intellectual prowess and engender co-

aware of themselves as part of the classroom social order. When students are overly concerned with establishing high status in relation to their peers, they can become outspoken, working hard to impress the teacher and each other. Others can also shut down, fearful that something they say may be evidence of their lack of knowledge compared to the rest of the class. Faculty may experi-

Learning goals for today

- Avoid premature optimization
- Know pitfalls of common APIs
- Understand garbage collection
- Ability to use a profiler

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.

—William A. Wulf

Competing Design Goals

- Extensibility
- Maintainability
  (design for change & understanding)
- Performance
- Safety, security
- Stability

"It is far, far easier to make a correct program fast than it is to make a fast program correct."

- Herb Sutter

Good Programs Rather than Fast Ones

- Information hiding:
  - Individual decisions can be changed and improved without affecting other parts of a system
  - Abstract interactions with the outside world (I/O, user interactions)
- A good architecture scales
- Hardware is cheap, developers are not
- Optimize only clear, concise, well-structured implementations, if at all
- Those who exchange readability for performance will lose both

Performance Optimizations

- High-level algorithmic changes

No amount of
low-level optimization
can fix an inefficient
algorithmic choice

- Low-level hacking

Before Optimization: **Profiling**

- Common wisdom: 80% of time spent in 20% of code
- Many optimizations have minimal impact or make performance worse
- Guessing problem often inefficient
- Use **profiler** to identify bottleneck
  - Often points toward algorithmic changes (quadratic -> linear)

EXAMPLE: COSINE SIMILARITY


Demo time

Performance prediction

- Performance prediction is hard
- Use profiler
- I/O can overshadow other costs
- Performance may not be practically relevant for many problems

15-313 Question

- Twitter famously had scalability problems and rewrote most of their system (Ruby -> Scala; Monolithic -> Microarchitecture)

- Was the initial monolithic design stupid?
- What tradeoffs to make for a startup?

PERFORMANCE PITFALLS (NOT ONLY IN JAVA)

Know the Language and its Libraries

- String concatenation
- List access
- Autoboxing

String concatenation in Java

```
public String toString(String[] elements) {
   String result = "";
   for (int i = 0; i < elements.length; i++)
       result += elements[i];
   return result;
}
```

String concatenation in Java

```
public String toString(String[] elements) {
  String result = "";
  for (int i = 0; i < elements.length; i++)
     result = result.concat(elements[i]);
  return result;
}
```

public String concat(String str)

"If the length of the argument string is **0**, then this **String** object is returned."

Efficient String Concatenation

```
public String toString(String[] elements) {
   StringBuilder b = new StringBuilder();
   for (int i = 0; i < elements.length; i++)
      b.append(elements[i]);
   return b.toString();
}
```

See implementation of StringBuilder

Lists

```
List<String> l = …
for (int i = 0; i < l.size(); i++)
      if ("key".equals(l.get(i))
            System.out.println("found it");
```

Possibly very slow; why?

Autoboxing: Integer vs int

- Integers are objects, ints are not
- new Integer(42) == new Integer(42) ?
- 4.equals(4) ?
- Integer a = 5 ?
- Math.max(12, new Integer(44)) ?
- new Integer(42) == 42 ?

see implementation of Integer

Understand Autoboxing

```
public static void main(String[] args) {
   Long sum = 0L;
   for (long i = 0; i < Integer.MAX_VALUE; i++) {
      sum += i;
   }
   System.out.println(sum);
}
```

Very slow; why?

When to use Boxed Primitives?

- Keys and values in collections (need objects)
- Type parameters in general (Optional<Long>)
- Prefer primitive types over boxed ones where possible

Understanding Hashcode

```
class Office {
		private String roomNr;
		private Set<Person> occupants;
		public boolean equals(Object that) { … }
}
Set<Office> …
```

possible problem?

Understanding Hashcode

```
class Office {
    private String roomNr;
    private Set<Person> occupants;
    public int hashCode() { return 0; }
}
Set<Office> …
```

performance problem?

Hashcode – good practice

- Start with nonzero constant (e.g. 17)
- For each significant field integrate value (result = result * 31 + c) where c:
  - "(f?1:0)" for boolean
  - "(int) f" for most primitives
  - o.hashCode for objects

Don’t worry about

- Overhead of method calls (e.g., strategy pattern)
- Overhead of object allocation (unless its millions)
- Multiplication vs shifting (compiler can optimize that)
- Performance of a single statement / microbenchmarks
- Recursion vs iteration

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

—Donald E. Knuth

GARBAGE COLLECTION

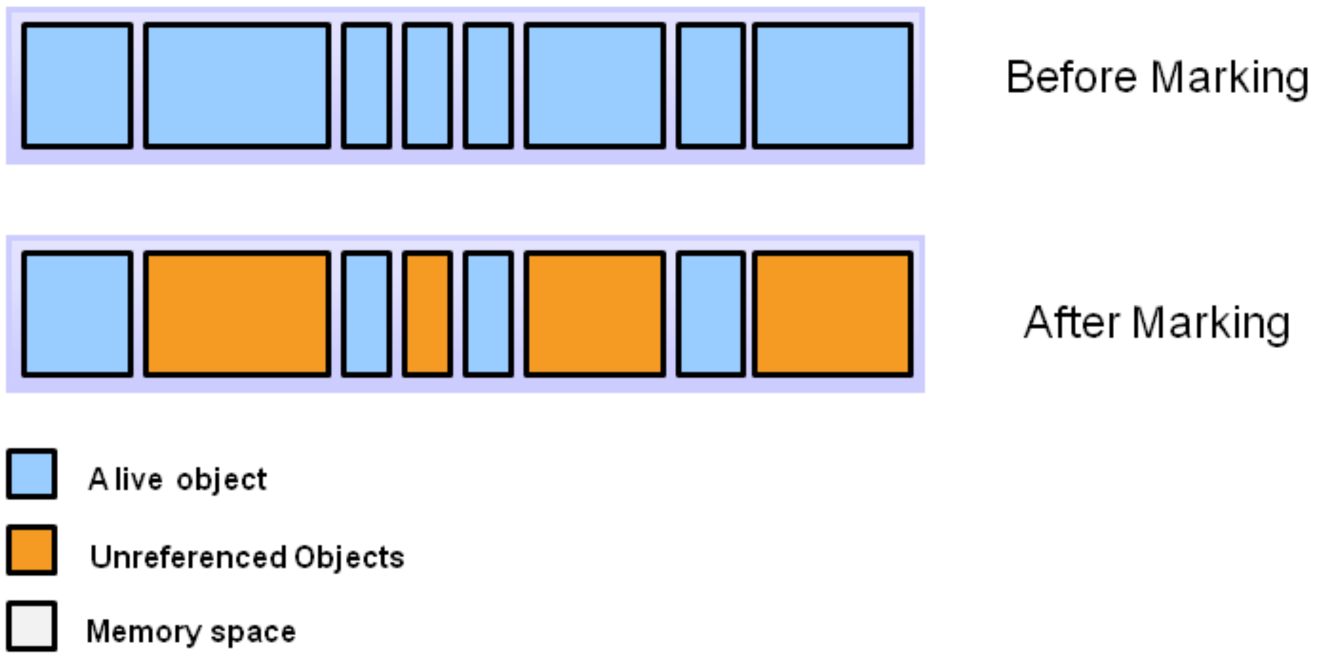
Explicit Memory Allocation vs. Garbage Collection

- Stack allocation:
  - `int x = 4;`
- Heap allocation
  - `Point x = new Point(4, 5);`
  - Reference on stack, object on heap
- C-style explicit memory allocation
  - `pointStruct* x; x = malloc(sizeof(pointStruct));`
  - `x -> y = 5; x -> x = 4;`
  - `free(x);`

Garbage Collection

- No explicit “free”
- Elements that are no longer referenced may be freed by the JVM
  - ```
    int foo() {
          Point x = new Point(4, 5);
          return x.x - x.y;
    }
    ```
  - ```
    set.add(new Point(4, 5));
    return set;
    ```

Marking



Before Marking
After Marking
A live object
Unreferenced Objects
Memory space

Memory Leaks

- C: Forgetting to free memory
- Java: Holding on to references to objects no longer needed
  - ```
    class Memory {
        static final List<Point> l = new ArrayList(10000);
        final HashMap<Integer, Connection> …
    }
    ```
- Java: Not closing streams, connections, etc
    ```

Memory Leak Example

```
class Stack {
    Point[] elements;
    int size = 0;
    void push(Point x) { elements[++size] = x; }
    Point peek() { return elements[size]; }
    Point pop() { return elements[size--]; }
}
```

Memory Leak Example

```
class Stack {
    …
    Point pop() {
        Point r = elements[size];
        elements[size] = null;
        size--;
        return r;
    }
}
```

Weak References

- References that may be garbage collected
  - java.lang.ref.WeakReference<T>
  - java.util.WeakHashMap<K,V> (weak keys)
- x = new WeakReference(new Point(4 ,5));
  x.get() // returns the point, or null if garbage collected in between
- WeakHashMap useful for caching, when cache should not prevent garbage collection

References and Observers

```
class Game {
    List<WeakReference<Listener>> listeners = …
    void addListener(Listener l) {
        listeners.add(new WeakReference(l));
    }
    void fireEvent() {
        for (wl : listeners) {
            Listener l = wl.get();
            if (l != null) l.update();
        }
    }
}
```

Should lists of observers be stored as weak references to avoid memory leaks?

Caching expensive computations (on immutable objects)

```
class Cache {
    Map<Cryptarithm, Solution> cache = new WeakHashMap<>();
    Solution solve(Cryptarithm c) {
        Solution result = cache.get(c);
        if (result != null) return result;
        result = c.solve();
        cache.put(c, result);
        return result;
    }
}
```

similar caching in factories when creating objects