Principles of Software Construction: Performance

Christian Kaestner Bogdan Vasilecu

School of Computer Science

institute for SOFTWARE RESEARCH

Your Feedback

- Recitations and homeworks useful
- Art vs performance
- Narrative of the class unclear
- Workload high, assignments too large
- Unclear how to act on feedback
- Suggestions:
  - More case studies of good design
  - Longer recitations
  - More live coding

Intro to Java

Git, CI

UML

Static Analysis

Performance

GUIs

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

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

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
- Who exchanges readability for performance will lose both

Performance Optimizations

- High-level algorithmic changes

- Low-level hacking

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

Performance informs design

- Find closest match in *n* documents
  - Computational complexity?
- Find closest matches in *n* documents
  - Computational complexity?
- What's the actual runtime performance?

Latency

PRIMITIVE	LATENCY: ns	us	ms
L1 cache reference	0.5		
Branch mispredict	5		
L2 cache reference	7		
Mutex lock/unlock	25		
Main memory reference	100		
Compress 1K bytes with Zippy	3,000	3	
Send 1K bytes over 1 Gbps network	10,000	10	
Read 4K randomly from SSD*	150,000	150	
Read 1 MB sequentially from memory	250,000	250	
Round trip within same datacenter	500,000	500	
Read 1 MB sequentially from SSD*	1,000,000	1,000	1
Disk seek	10,000,000	10,000	10
Read 1 MB sequentially from disk	20,000,000	20,000	20
Send packet CA->Netherlands->CA	150,000,000	150,000	150

```
public class Document {
    private final ...

    public Document(String url) throws IOException {


    }

    public double cosineSimilarity(Document doc) {


    }
}
```

(redacted)

(redacted)

Profiler Demo

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?

Scrabble Design

- When to load the dictionary?
- When to check whether a move is valid?

PERFORMANCE PITFALLS (NOT ONLY IN JAVA)

Know the Language and its Libraries

- String concatenation
- List access
- Autoboxing
- Hashcode

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;
}
```

See implementation of String.concat()

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

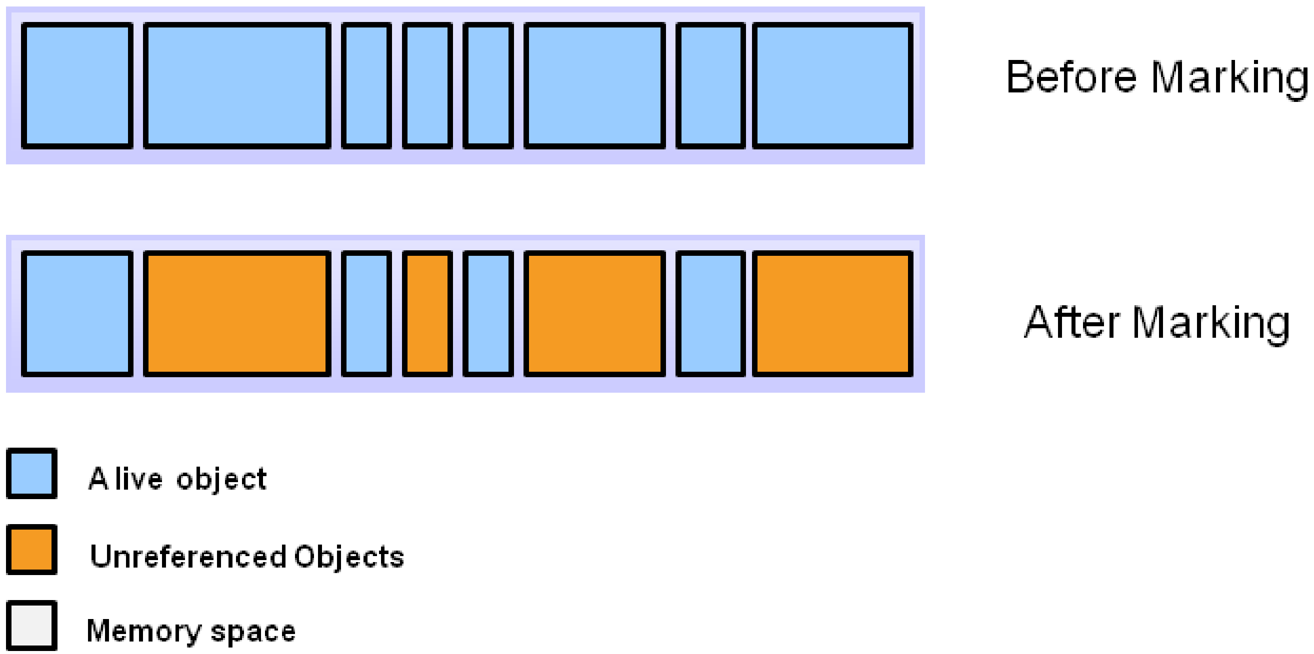
Explicit Memory Allocation vs. Garbage Collection

- 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() {
    ```
    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--]; }
}
```

Why is this a problem? How to fix it?

institute for SOFTWARE RESEARCH

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

PERFORMANCE AND DESIGN

Performance in API Design

- Immutable classes are easy and fast
  - Easy to share
  - No defensive copying
- class type instead of interface type ties to that class; inheritance ties subclass to superclass decisions, delegation does not

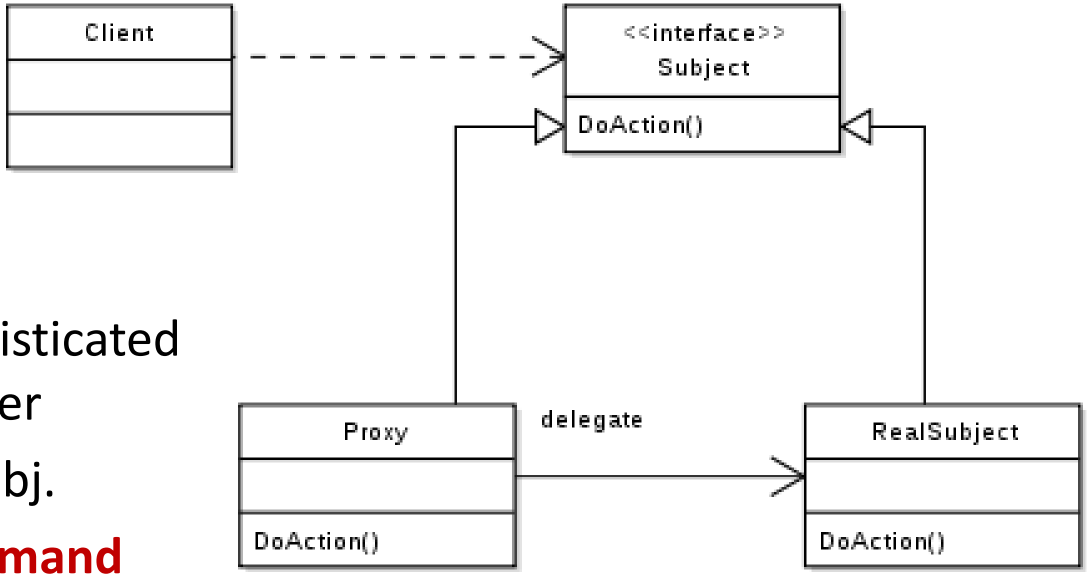
Example: Poor Performance through API Design

- java.awt.Component.getSize returns mutable Dimension
  - lots of defensive copying
  - separate getWidth/getHight methods added later for performance reasons
    - “Returns the current height of this component. This method is preferable to writing component. getBounds().height or component.getSize().height because it doesn't cause any heap allocations.”
- Old design problems stick around

Design Pattern for Performance

- Flyweight
- Proxy (caching)
- Factories (caching)

Proxy Design Pattern



Applicability

- Whenever you need a more sophisticated obj reference than a simple pointer
- Local representative for remote obj.
- **Create/load expensive obj on demand**
- Control access to an object
- Extra error handling, failover
- **Caching**
- Reference count an object

Consequences

- Introduces a level of indirection
- Hides distribution from client
- Hides optimizations from client
- Adds housekeeping tasks

Proxy Example

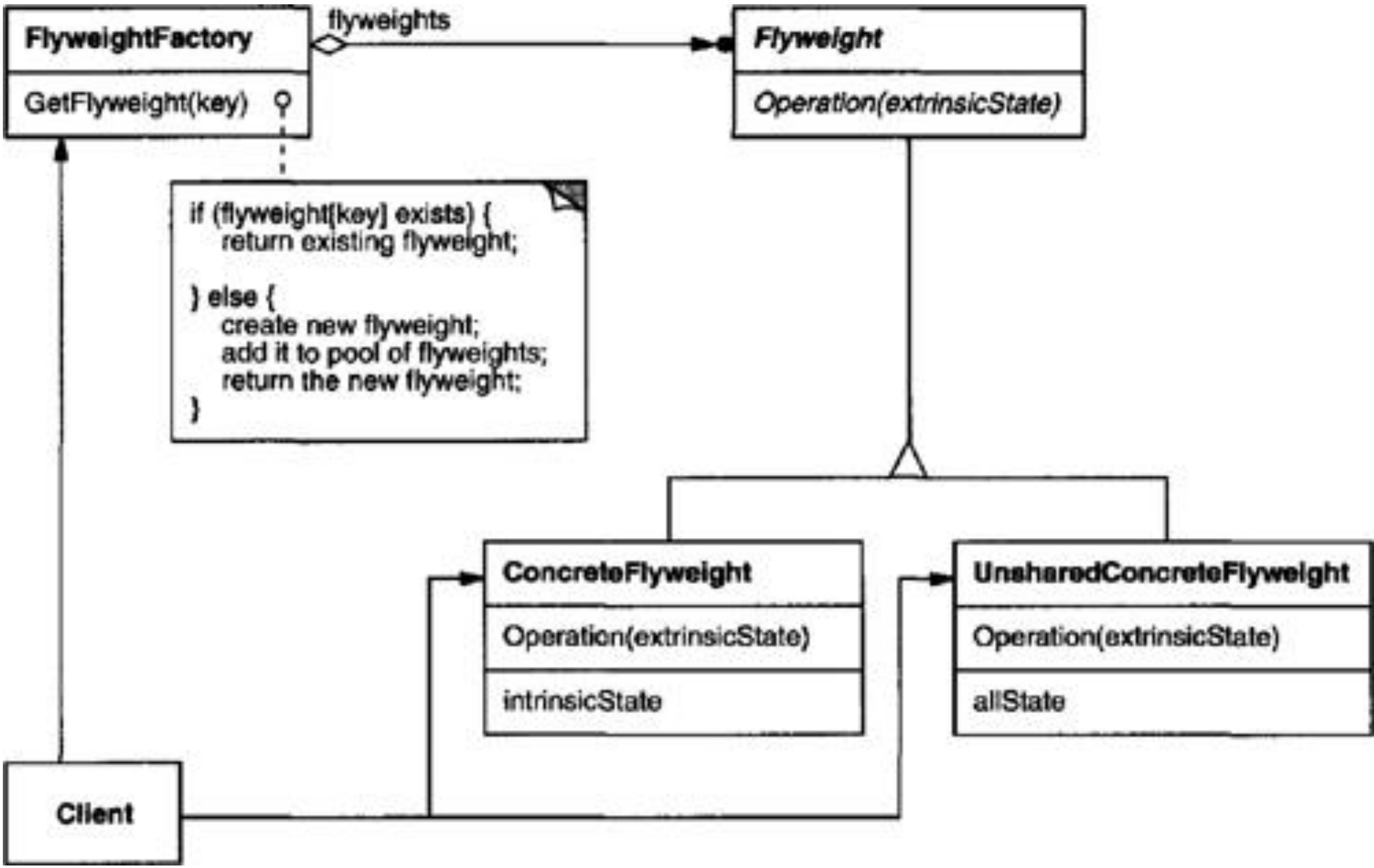
```
CryptarythmProxy implements Cryptarythm {
    private Cryptarythm c;
    private final String[] input;
    CryptarythmProxy(String[] words) { input = words; }
    public solve() {
        if (c != null)
            c = new Cryptarythm(input);
        return c.solve();
    }
}
```

Proxy Example

```
CryptarythmProxy implements Cryptarythm {
    private Solution solution;
    private final String[] input;
    CryptarythmProxy(String[] words) { input = words; }
    public solve() {
        if (solution != null)
            solution = new Cryptarythm(input).solve();
        return solution;
    }
}
```

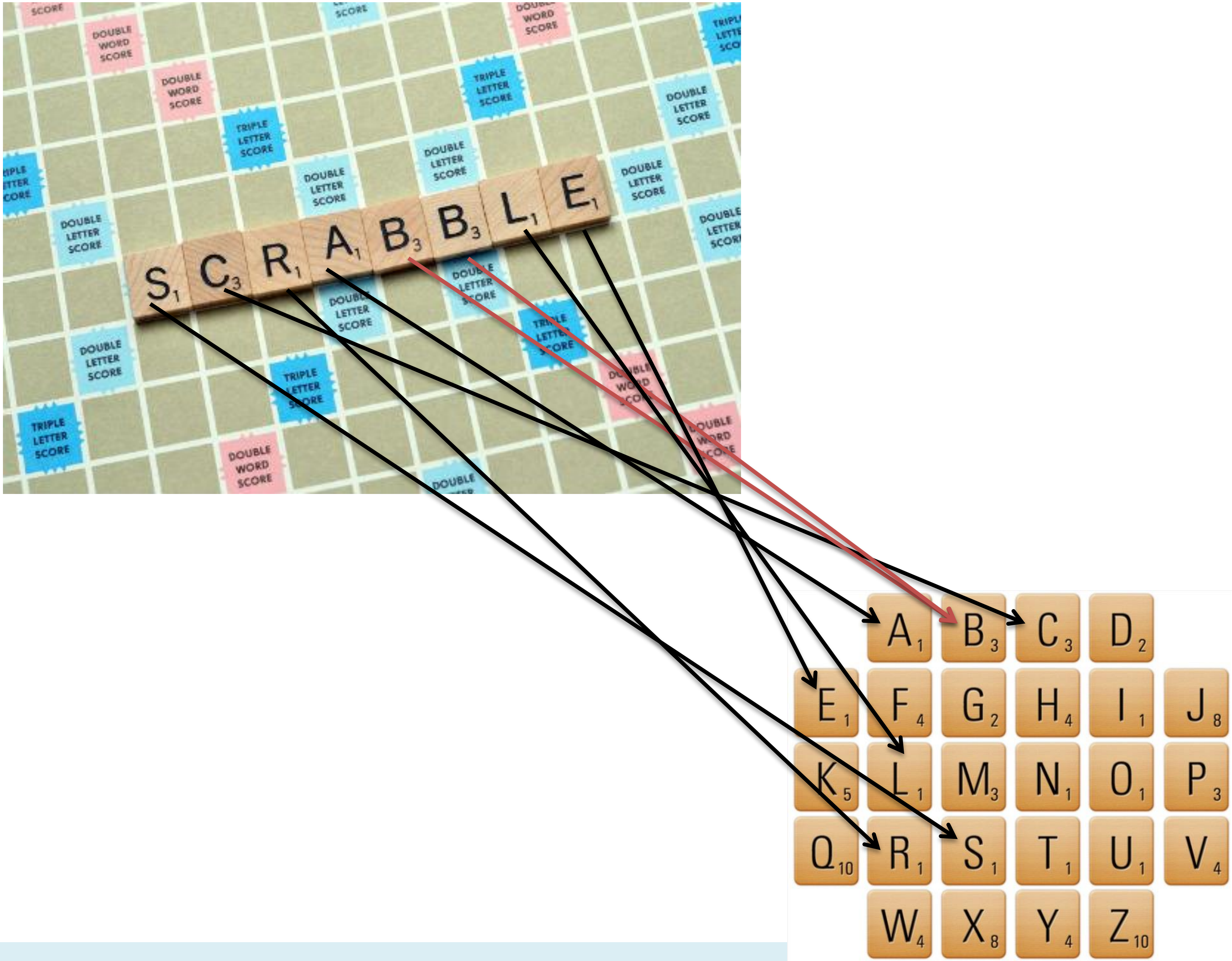
The Flyweight Pattern

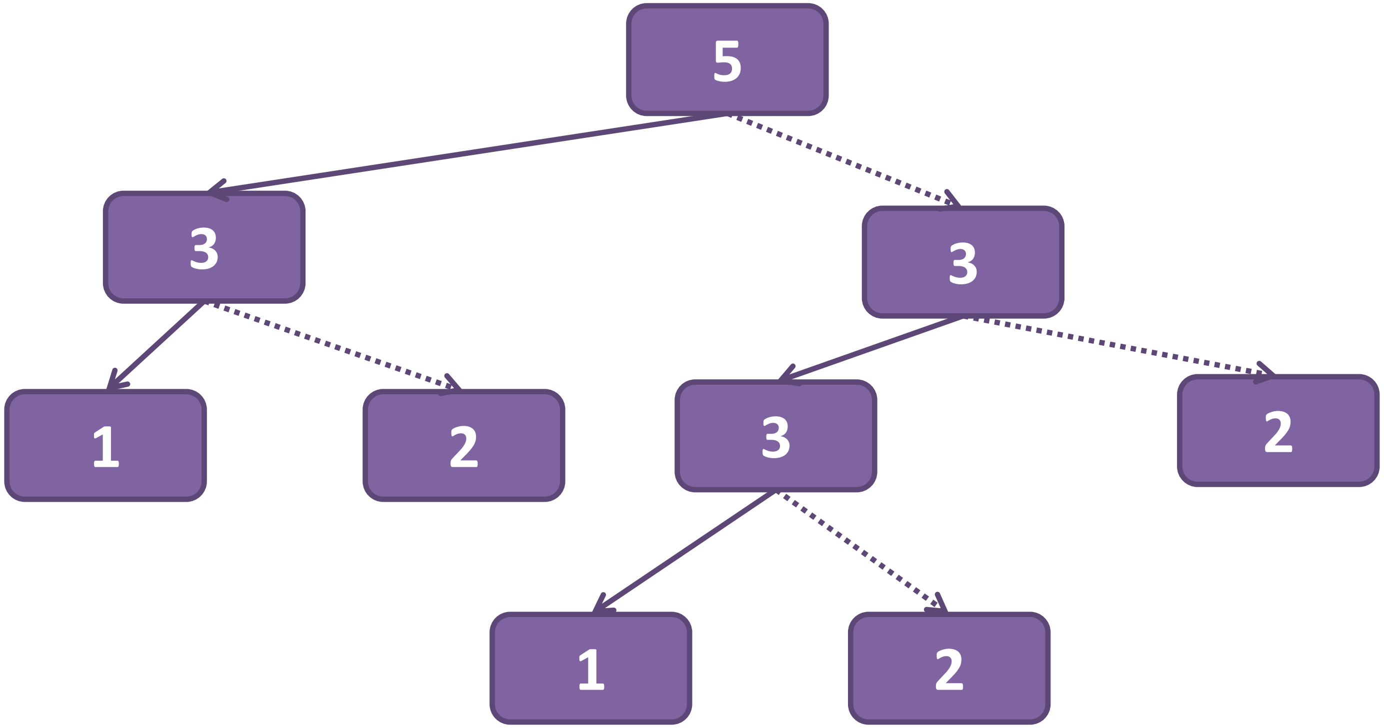
- Share data structures for values efficiently; create one instance per value
- Examples:
  - Characters in a document
  - Enums
  - Coffee Flavors
- Flyweights are immutable value objects, their creation is cached in a factory
- Aka “Hash consing”


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

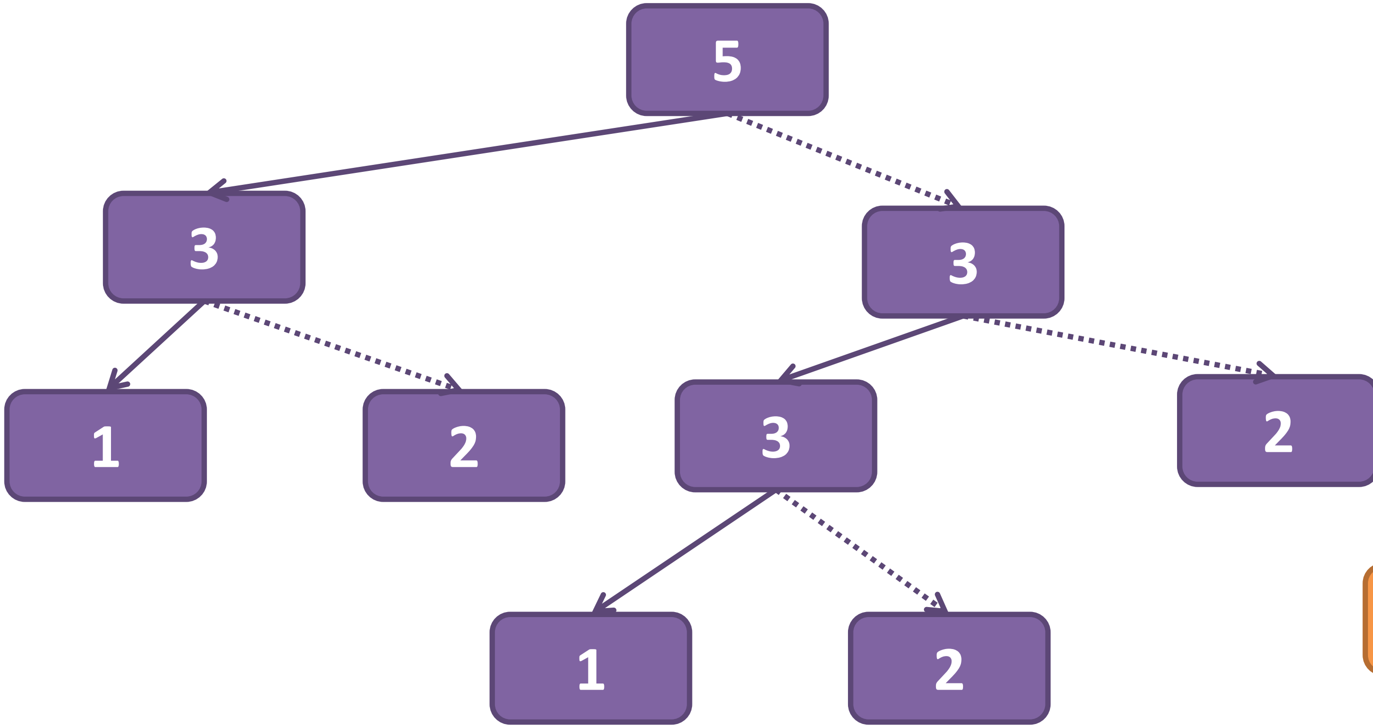
Flyweight Example

```
class TileImage { // immutable value class, the flyweight
   TileImage(char c) { … } // package-visible constructor can prevent
   image, draw() …          // clients from instantiating directly
}
class TileImageFactory {
    private Map<Char, TileImage> cache = new WeakHashMap<>();
    public TileImage create(char c) {
      TileImage result = cache.get(c);
      if (result != null) return result;
      result = new TileImage(c);
      cache.put(c, result);
      return result;
   }
}
```

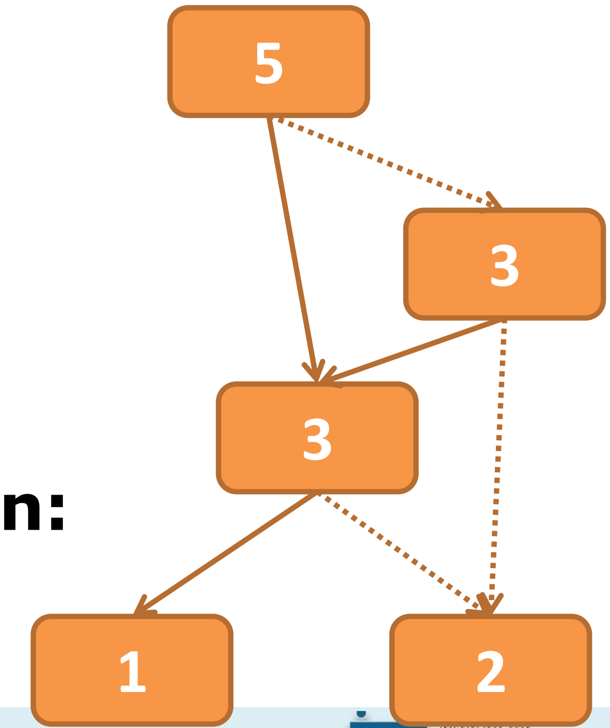

S C R A B B L E
DOUBLE WORD SCORE
TRIPLE LETTER SCORE
DOUBLE LETTER SCORE
A B C D
E F G H I J
K L M N O P
Q R S T U V
W X Y Z



How can we represent the same tree with fewer objects?



Reusing Tree Nodes with Flyweight Pattern:



Conclusion

- Performance does not matter, until it does
- Focus on good designs, avoid premature optimization
- Use a profiler before optimizing
- Know pitfalls in Java, understand weak references
- Flyweight, Proxies, *Factory Patterns all enable caching of sorts

Further Reading

- Effective Java, Item 55 and many more
- Design patterns Proxy, Flyweight, *Factory
- Java API documentation of WeakReference, WeakHashmap