Formal Specification and Verification of JDK's Identity Hash Map Implementation

Martin de Boer, **Stijn de Gouw**, Jonas Klamroth,
Christian Jung, Mattias Ulbrich, Alexander Weigl

Open Universiteit
Faculteit Management, Science & Technology

API: core methods

Map: collection of key-value pairs, where keys are unique
(similar to functions in mathematics)

containsKey(**Object** key)
Tests whether the specified object reference is a key in this identity hash map.

V get(**Object** key)
Returns the value to which the specified key is mapped,
or null if this map contains no mapping for the key.

V put(**K** key, **V** value)
Associates the specified value with the specified key in this identity hash map.

V remove(**Object** key)
Removes the mapping for this key from this map if present.

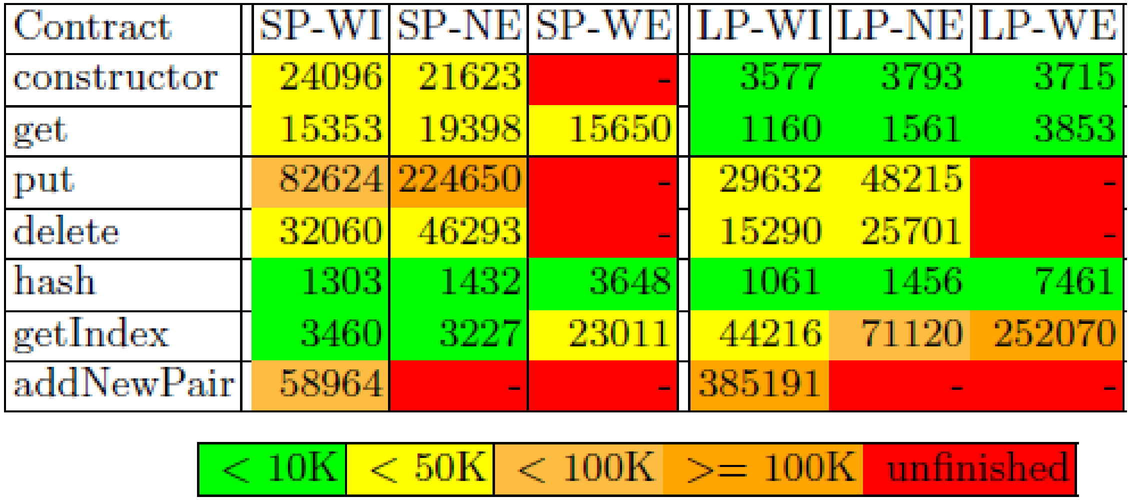
IdentityHashMap

- Real-world hash map implementation, part of the Java Collection Framework
- No known previous formal verification attempts
- Complex class, but simpler than other HashMaps
  - State space / representation simpler than other HashMap implementations: all entries are stored in a single array
  - Compares keys with `==' rather than equals method
  - Integer overflow semantics exploited, bitwise operations
  - > 1200 LoC (incl. comments / white space)
  - Challenge: analyse (nearly) unaltered code

Idealized hash maps

Study by Christian Jung with maps optimized for verif

- SP-…: collision resolution with separate chaining
- LP-…: collision resolution with linear probing
- …-WI: keys are ints (compared with ==)
- …-NE: keys are objects compared with ==
- …-WE: separate chaining with objects and equals(..)

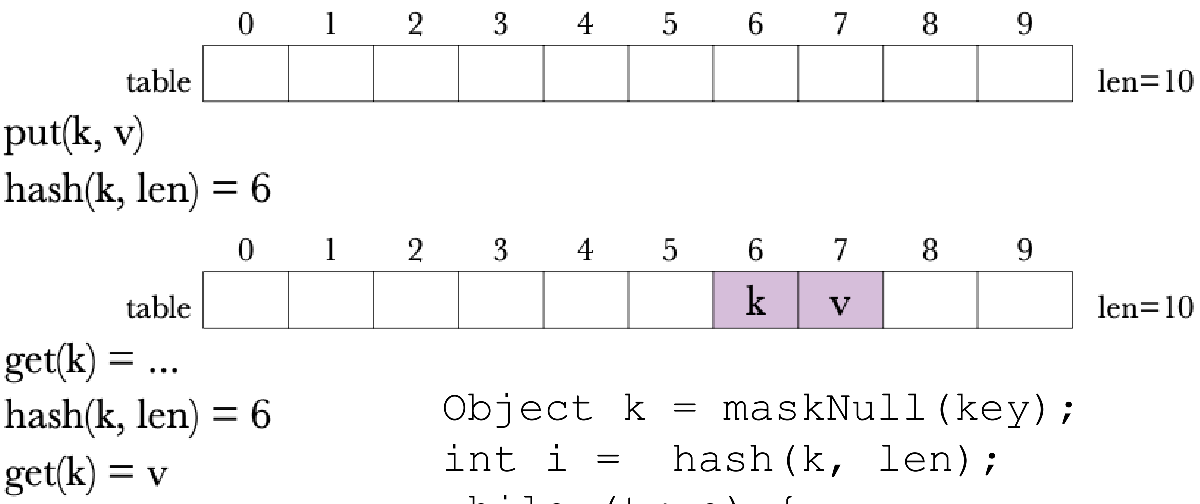


Contract	SP-WI	SP-NE	SP-WE	LP-WI	LP-NE	LP-WE
constructor	24096	21623	-	3577	3793	3715
get	15353	19398	15650	1160	1561	3853
put	82624	224650	-	29632	48215	-
delete	32060	46293	-	15290	25701	-
hash	1303	1432	3648	1061	1456	7461
getIndex	3460	3227	23011	44216	71120	252070
addNewPair	58964	-	-	385191	-	-

🟩 < 10K	🟨 < 50K	🟧 < 100K	🟧 >= 100K	🟥 unfinished

hash, put, get

- Key unicity based on reference equality `=='
- Hash table: key stored at even index determined by hash function, values at (next) odd index



```
Object k = maskNull(key);
int i =  hash(k, len);
while (true) {
    Object item = tab[i];
    if (item == k)
        return (V) tab[i + 1];
    if (item == null)
        return null;
    i = nextKeyIndex(i, len);
}
```

Collision resolution

- Good hash is often unique, but no guarantees
- Different key, same hash: collision
- Resolve collisions with linear probing

	0	1	2	3	4	5	6	7	8	9	
table							k	v			len=10

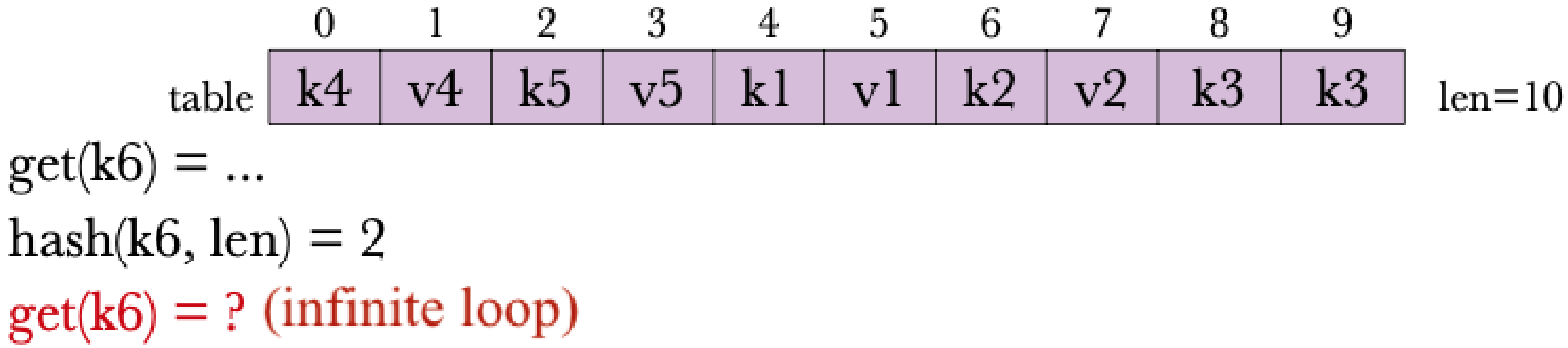
put(k2, v2)
hash(k2, len) = 6
put(k3, v3)
hash(k3, len) = 6

	0	1	2	3	4	5	6	7	8	9	
table	k3	v3					k	v	k2	v2	len=10

get(k3) = ...
hash(k3, len) = 6
get(k3) = v3

Empty slot

Hash table must never get fully occupied



	0	1	2	3	4	5	6	7	8	9	
table	k4	v4	k5	v5	k1	v1	k2	v2	k3	k3	len=10

get(k6) = ...

hash(k6, len) = 2

get(k6) = ? (infinite loop)

```
Object k = maskNull(key);
int i =  hash(k, len);
while (true) {
    Object item = tab[i];
    if (item == k)
        return (V) tab[i + 1];
    if (item == null)
        return null;
    i = nextKeyIndex(i, len);
}
```

remove

	0	1	2	3	4	5	6	7	8	9	
table	k3	v3					k	v	k2	v2	len=10

remove(k)

	0	1	2	3	4	5	6	7	8	9	
table	k3	v3							k2	v2	len=10

get(k2) = ...

ERROR

hash(k2, len) = 6

get(k2) = null

remove (2)

	0	1	2	3	4	5	6	7	8	9	
table	k3	v3					k	v	k2	v2	len=10

remove(k)

	0	1	2	3	4	5	6	7	8	9	
table							k2	v2	k3	k3	len=10
							←	←	←	←	

get(k2) = ...

hash(k2, len) = 6

get(k2) = v2

Specification: Class invariant

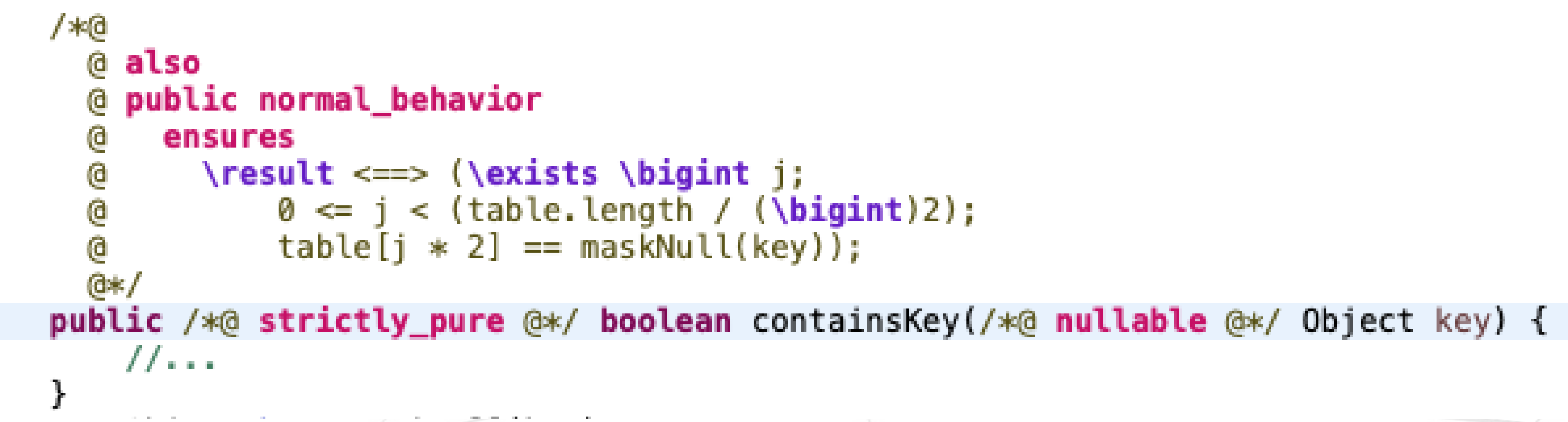
- Main conditions:
  - Keys must be unique (reference equality)
  - No gaps (empty slots) between keys with identical hashes
  - Table must at all times have at least one empty entry
  - …

Class invariant (2)

```
/*@
  @ ...
  @
  @ // Non-empty keys are unique
  @ public invariant
  @   (\forall \bigint i; 0 <= i && i < table.length / (\bigint)2;
  @       (\forall \bigint j;
  @       i <= j && j < table.length / (\bigint)2;
  @       (table[2 * i] != null && table[2 * i] == table[2 * j]) ==> i == j));
  @
  @ ...
  @
  @ // Table must have at least one empty key-element to prevent
  @ // infinite loops when a key is not present.
  @ public invariant
  @   (\exists \bigint i;
  @       0 <= i < table.length / (\bigint)2;
  @       table[2 * i] == null);
  @
  @ // There are no gaps between a key's hashed index and its actual
  @ // index (if the key is at a higher index than the hash code)
  @ public invariant
  @   (\forall \bigint i;
  @       0 <= i < table.length / (\bigint)2;
  @       table[2 * i] != null && 2 * i > hash(table[2 * i], table.length) ==>
  @       (\forall \bigint j;
  @           hash(table[2 * i], table.length) / (\bigint)2 <= j < i;
  @           table[2 * j] != null));
  @
  @ // There are no gaps between a key's hashed index and its actual
  @ // index (if the key is at a lower index than the hash code)
  @ public invariant
  @   (\forall \bigint i;
  @       0 <= i < table.length / (\bigint)2;
  @       table[2 * i] != null && 2 * i < hash(table[2 * i], table.length) ==>
  @       (\forall \bigint j;
  @           hash(table[2 * i], table.length) <= 2 * j < table.length || 0 <= 2 * j < 2 * i;
  @           table[2 * j] != null));
  @
  @ ...
  @
  @*/
```

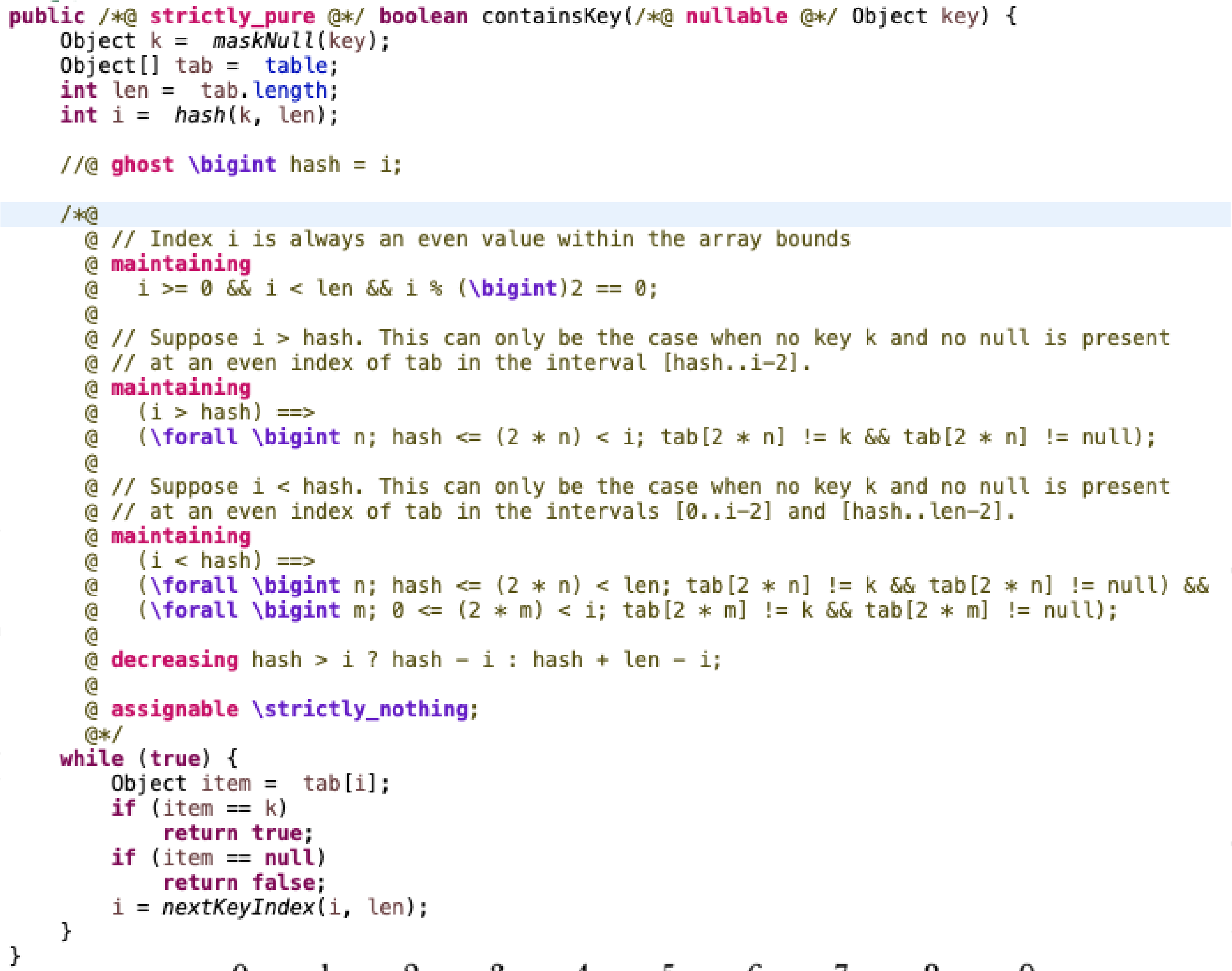
Proof containsKey

- Method preserves class invariant (trivial)
- Method satisfies below JML-contract

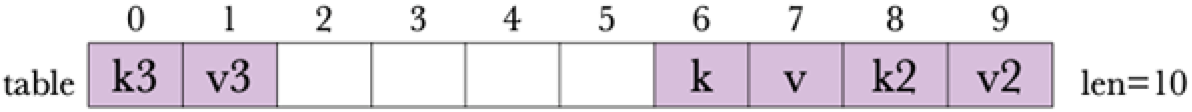


```
/*@
  @ also
  @ public normal_behavior
  @   ensures
  @     \result <==> (\exists \bigint j;
  @         0 <= j < (table.length / (\bigint)2);
  @         table[j * 2] == maskNull(key));
  @*/
public /*@ strictly_pure @*/ boolean containsKey(/*@ nullable @*/ Object key) {
    //...
}
```

Proof containsKey (3)



```
public /*@ strictly_pure @*/ boolean containsKey(/*@ nullable @*/ Object key) {
    Object k =  maskNull(key);
    Object[] tab =  table;
    int len =  tab.length;
    int i =  hash(k, len);

    //@ ghost \bigint hash = i;

    /*@
      @ // Index i is always an even value within the array bounds
      @ maintaining
      @   i >= 0 && i < len && i % (\bigint)2 == 0;
      @
      @ // Suppose i > hash. This can only be the case when no key k and no null is present
      @ // at an even index of tab in the interval [hash..i-2].
      @ maintaining
      @   (i > hash) ==>
      @   (\forall \bigint n; hash <= (2 * n) < i; tab[2 * n] != k && tab[2 * n] != null);
      @
      @ // Suppose i < hash. This can only be the case when no key k and no null is present
      @ // at an even index of tab in the intervals [0..i-2] and [hash..len-2].
      @ maintaining
      @   (i < hash) ==>
      @   (\forall \bigint n; hash <= (2 * n) < len; tab[2 * n] != k && tab[2 * n] != null) &&
      @   (\forall \bigint m; 0 <= (2 * m) < i; tab[2 * m] != k && tab[2 * m] != null);
      @
      @ decreasing hash > i ? hash - i : hash + len - i;
      @
      @ assignable \strictly_nothing;
      @*/
    while (true) {
        Object item =  tab[i];
        if (item == k)
            return true;
        if (item == null)
            return false;
        i = nextKeyIndex(i, len);
    }
}
```



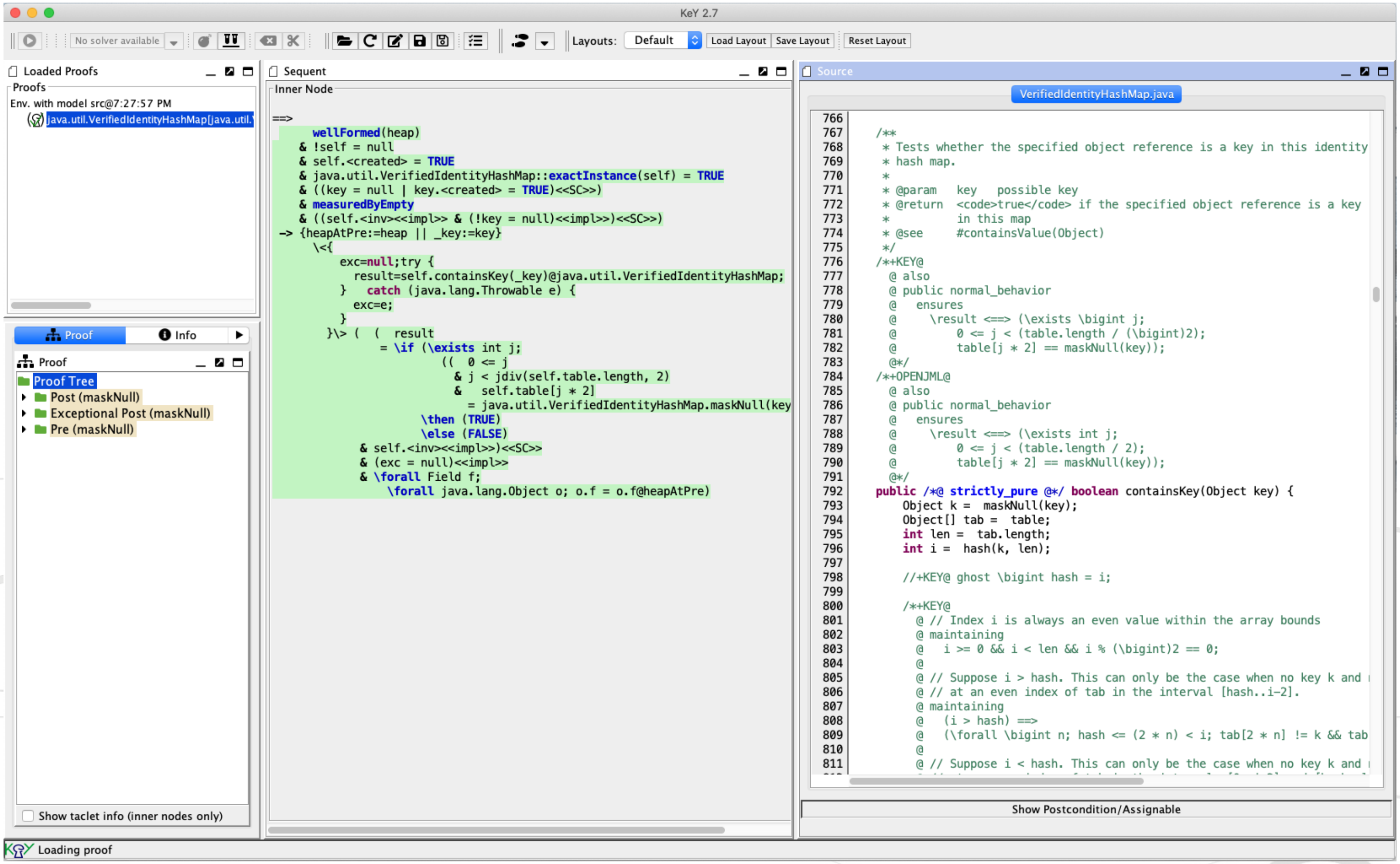
	0	1	2	3	4	5	6	7	8	9	
table	k3	v3					k	v	k2	v2	len=10

Proof containsKey (2)

Loop termination

- First phase: hash $\leq$ i $\leq$ len-2. Clearly hash+len-i is positive. If i=len-2, then hash+len-i = hash+2
- Second phase, after wraparound: 0 $\leq$ i < hash. If i=0 then hash-i = hash (i.e. >0 and decreasing) And if i increases then hash-i decreases
- Furthermore, i cannot become equal to hash since all keys are then != null according to the loop inv, while the class inv implies there must be a null (at an even index).

Proof containsKey (4)



```
==>
      wellFormed(heap)
   & !self = null
   & self.<created> = TRUE
   & java.util.VerifiedIdentityHashMap::exactInstance(self) = TRUE
   & ((key = null | key.<created> = TRUE)<<SC>>)
   & measuredByEmpty
   & ((self.<inv><<impl>> & (!key = null)<<impl>>)<<SC>>)
 -> {heapAtPre:=heap || _key:=key}
      \<{
          exc=null;try {
            result=self.containsKey(_key)@java.util.VerifiedIdentityHashMap;
          }   catch (java.lang.Throwable e) {
            exc=e;
          }
        }\> (  (  result
                = \if (\exists int j;
                         ((  0 <= j
                           & j < jdiv(self.table.length, 2)
                           &   self.table[j * 2]
                             = java.util.VerifiedIdentityHashMap.maskNull(key
                    \then (TRUE)
                    \else (FALSE)
             & self.<inv><<impl>>)<<SC>>
             & (exc = null)<<impl>>
             & \forall Field f;
                 \forall java.lang.Object o; o.f = o.f@heapAtPre)
```

```

    /**
     * Tests whether the specified object reference is a key in this identity
     * hash map.
     *
     * @param   key   possible key
     * @return  <code>true</code> if the specified object reference is a key
     *          in this map
     * @see     #containsValue(Object)
     */
    /*+KEY@
      @ also
      @ public normal_behavior
      @   ensures
      @     \result <==> (\exists \bigint j;
      @         0 <= j < (table.length / (\bigint)2);
      @         table[j * 2] == maskNull(key));
      @*/
    /*+OPENJML@
      @ also
      @ public normal_behavior
      @   ensures
      @     \result <==> (\exists int j;
      @         0 <= j < (table.length / 2);
      @         table[j * 2] == maskNull(key));
      @*/
    public /*@ strictly_pure @*/ boolean containsKey(Object key) {
        Object k =  maskNull(key);
        Object[] tab =  table;
        int len =  tab.length;
        int i =  hash(k, len);

        //+KEY@ ghost \bigint hash = i;

        /*+KEY@
          @ // Index i is always an even value within the array bounds
          @ maintaining
          @   i >= 0 && i < len && i % (\bigint)2 == 0;
          @
          @ // Suppose i > hash. This can only be the case when no key k and
          @ // at an even index of tab in the interval [hash..i-2].
          @ maintaining
          @   (i > hash) ==>
          @   (\forall \bigint n; hash <= (2 * n) < i; tab[2 * n] != k && tab
          @
          @ // Suppose i < hash. This can only be the case when no key k and
```

Hybrid analysis

Main goal: decreasing the effort of formal analysis

- Small change in specs, such as class invariant, typically break (re-)loading existing partial proof early
    - Currently ongoing experiment: use proof scripts
- Main bottleneck: writing good (correct, sufficient) specs

Hybrid approach

Early detection of specification errors

- JMLUnit/JMLUnitNG ▯ not maintained anymore, we aborted after (too) much effort needed to load case study
- OpenJML ▯ lib too complex but did discover syntactic / visibility errors in specs (more sensitive than KeY)
- JUnit/Reflection (unit tests)
- JJBMC (model checker, Jonas Klamroth (KIT/FZI))

JUnit/Reflection

Strong points

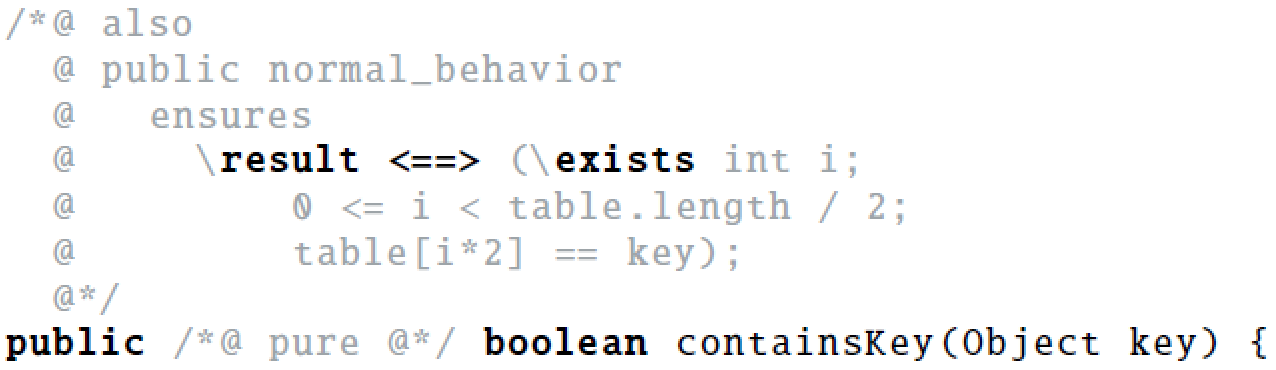
- Detection of semantic errors in specs
- Reflection provides access to a class's inner state
- When carefully designed, re-use of automatic testing of the class-invariant is possible for all methods

Limitations

- 'Manual' translation of JML to Java
    - False positives
    - False negatives
- Not suitable for loop invariants and block contracts
- Extra maintenance during analysis process

JJBMC: strong points

- Good developer Improved tool quickly based on case study
- Fully automatic, limited time needed to load case study
- Early detection of errors in several method specs, including discovery of non-trivial semantic errors



```
/*@ also
  @ public normal_behavior
  @   ensures
  @     \result <==> (\exists int i;
  @         0 <= i < table.length / 2;
  @         table[i*2] == key);
  @*/
public /*@ pure @*/ boolean containsKey(Object key) {
```

- Can identify whether specs are insufficient
- Outputs counter-example

JJBMC: limitations

Future work?

- State space explosion (capacity upper-bound of 4 entries)
- Limitations in OpenJML dialect (e.g. exceptional behaviour not supported, bsum, bprod, loop invariants)
- No support for functions without Java method body
  - user-defined functions/predicates from a .key file
  - calls to native code
- Limitations wrt aliasing (diff vars cannot alias)

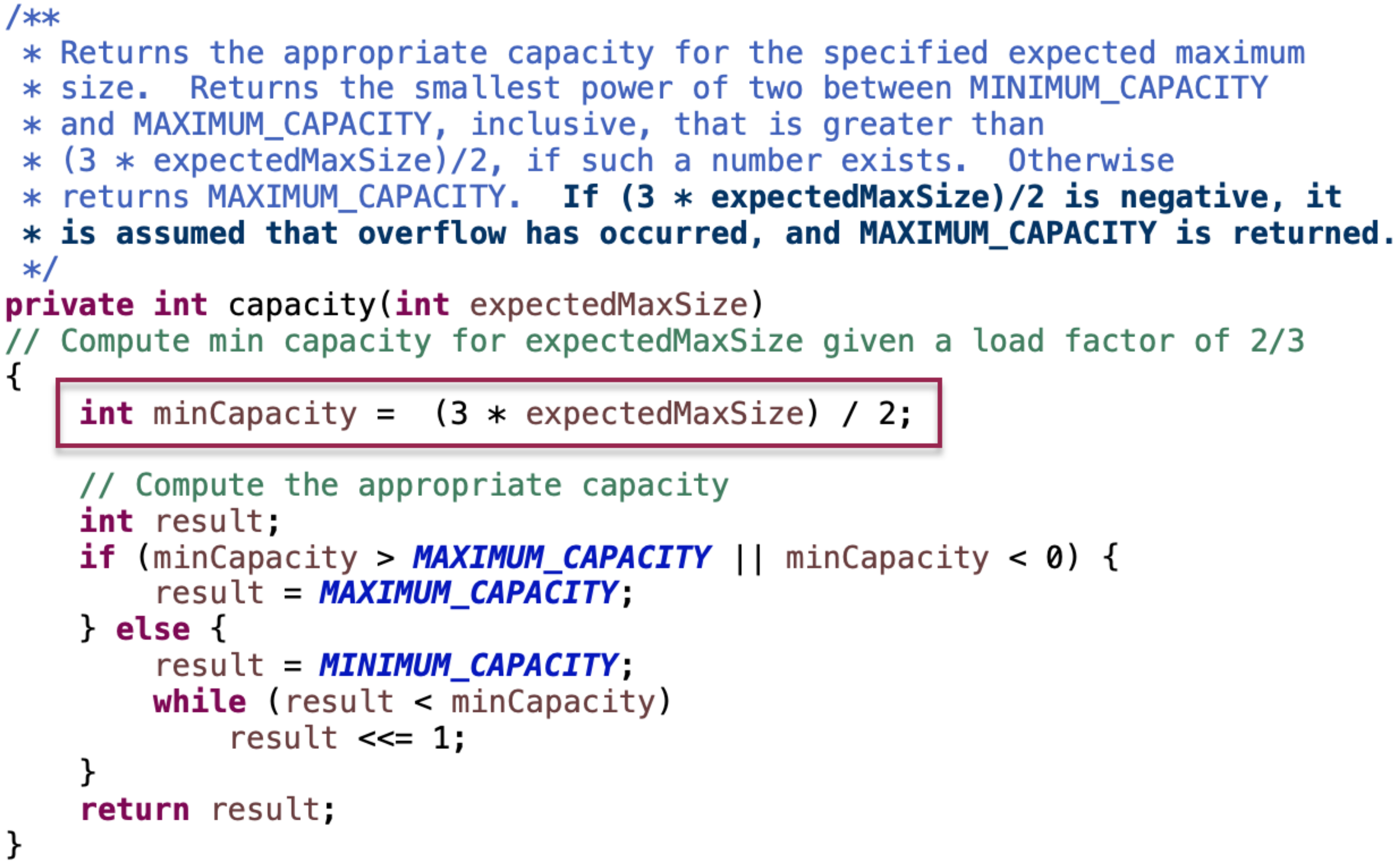
Effort ratio

- Difficult to measure: how to compare with and without hybrid analysis?
- Rough estimate from student based on planned hours: efficiency improved with about 12,54% due to hybrid approach (junit tests and JJBMC)

Discovered bugs?

How to fix?

Capacity



```
/**
 * Returns the appropriate capacity for the specified expected maximum
 * size.  Returns the smallest power of two between MINIMUM_CAPACITY
 * and MAXIMUM_CAPACITY, inclusive, that is greater than
 * (3 * expectedMaxSize)/2, if such a number exists.  Otherwise
 * returns MAXIMUM_CAPACITY.  If (3 * expectedMaxSize)/2 is negative, it
 * is assumed that overflow has occurred, and MAXIMUM_CAPACITY is returned.
 */
private int capacity(int expectedMaxSize)
// Compute min capacity for expectedMaxSize given a load factor of 2/3
{
    int minCapacity =  (3 * expectedMaxSize) / 2;

    // Compute the appropriate capacity
    int result;
    if (minCapacity > MAXIMUM_CAPACITY || minCapacity < 0) {
        result = MAXIMUM_CAPACITY;
    } else {
        result = MINIMUM_CAPACITY;
        while (result < minCapacity)
            result <<= 1;
    }
    return result;
}
```

MAXIMUM_CAPACITY = 1 << 29 = 2^{29} = 536.870.912

MINIMUM_CAPACITY = 4

Capacity error

(3 * 1431655766) / 2 = 1 ✗

capacity(1431655766) = 4 ✗ (expected: 536870912)

(3 * 1431655772) / 2 = 10 ✗

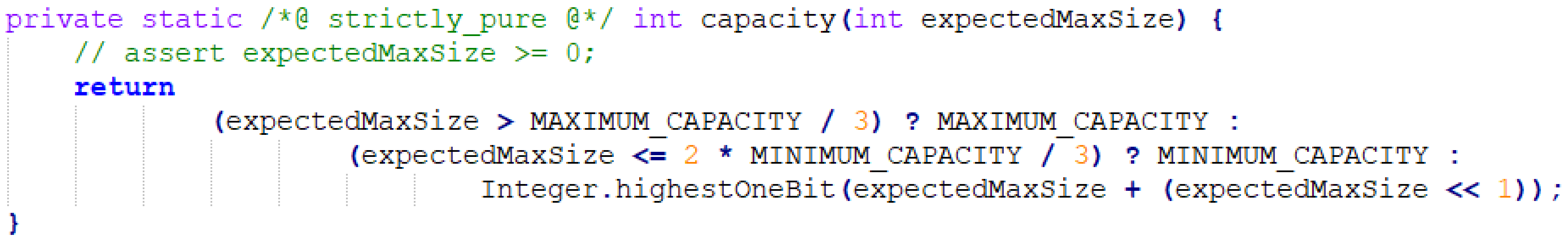
capacity(1431655772) = 16 ✗ (expected: 536870912)

Error is triggered in range 1.431.655.765 – 1.610.612.736

Consequences of undetected overflow

- Table allocated with *far* too little capacity
- Main purpose of constructor with expected max size: increased performance
- Many resizes when putting new entries in table: entries shuffled to other positions due to recalculated hashes
- Performance declined by about 45%

New capacity in later JDK update



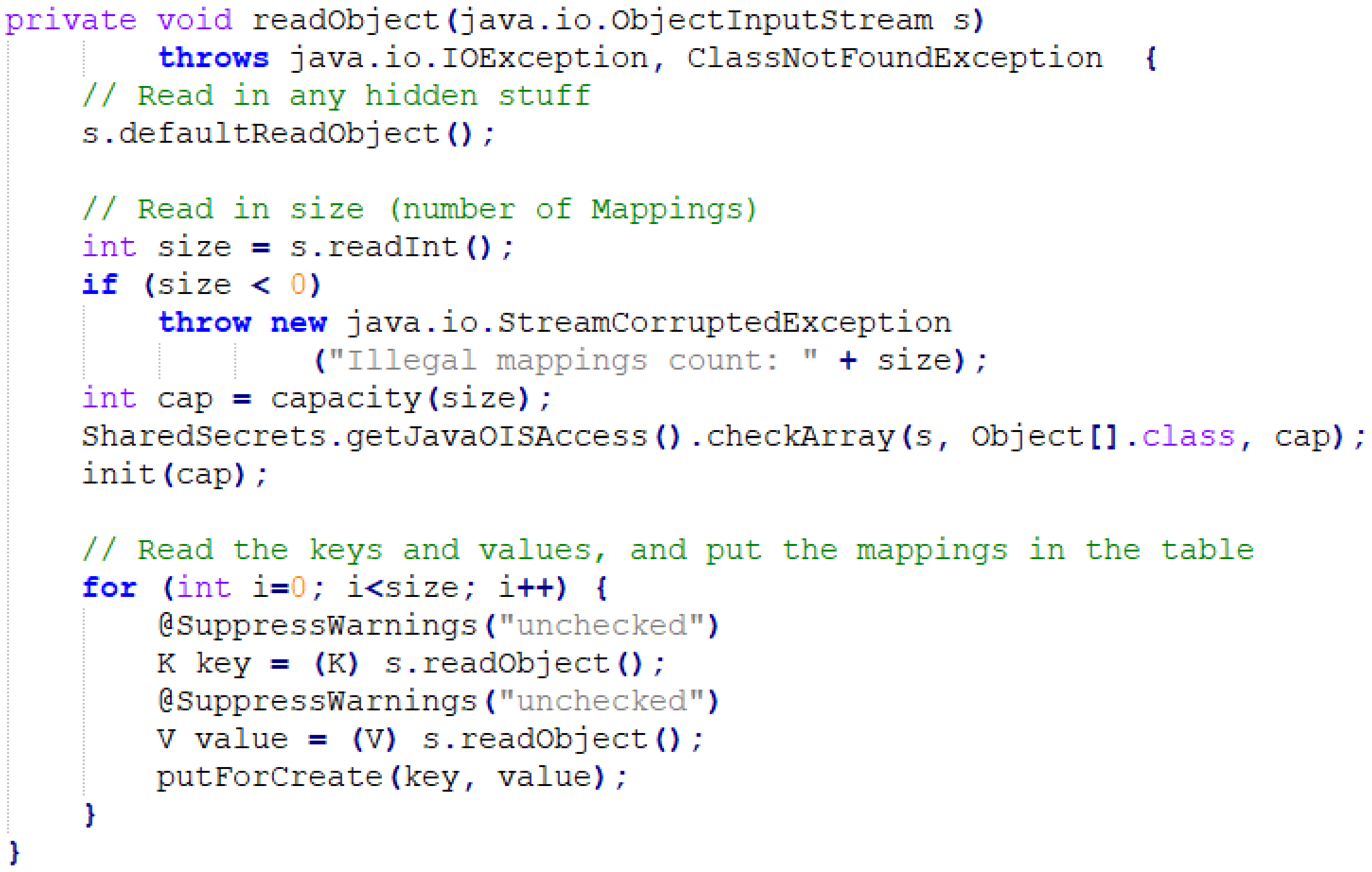
```
private static /*@ strictly_pure @*/ int capacity(int expectedMaxSize) {
    // assert expectedMaxSize >= 0;
    return
            (expectedMaxSize > MAXIMUM_CAPACITY / 3) ? MAXIMUM_CAPACITY :
                    (expectedMaxSize <= 2 * MINIMUM_CAPACITY / 3) ? MINIMUM_CAPACITY :
                            Integer.highestOneBit(expectedMaxSize + (expectedMaxSize << 1));
}
```

MAXIMUM_CAPACITY = 1 << 29 = 2^{29} = 536870912

MINIMUM_CAPACITY = 4

Serialization: readObject

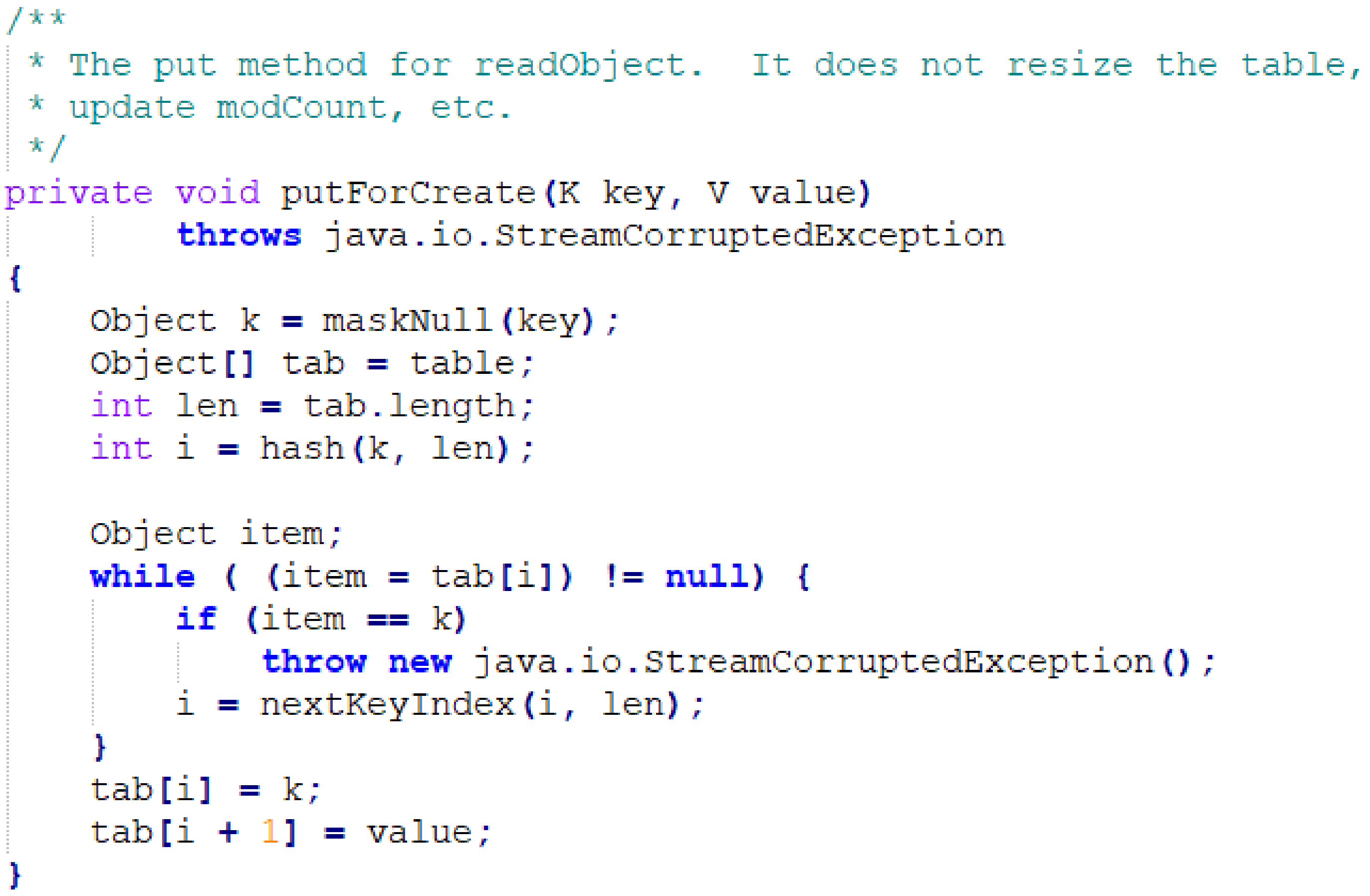
Used for serialization: writing an IdentityHashMap object + contents to a stream (e.g. a file)



```
private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException  {
    // Read in any hidden stuff
    s.defaultReadObject();

    // Read in size (number of Mappings)
    int size = s.readInt();
    if (size < 0)
        throw new java.io.StreamCorruptedException
                ("Illegal mappings count: " + size);
    int cap = capacity(size);
    SharedSecrets.getJavaOISAccess().checkArray(s, Object[].class, cap);
    init(cap);

    // Read the keys and values, and put the mappings in the table
    for (int i=0; i<size; i++) {
        @SuppressWarnings("unchecked")
        K key = (K) s.readObject();
        @SuppressWarnings("unchecked")
        V value = (V) s.readObject();
        putForCreate(key, value);
    }
}
```

Serialization: readObject

Note: no resize!



```java
/**
 * The put method for readObject.  It does not resize the table,
 * update modCount, etc.
 */
private void putForCreate(K key, V value)
        throws java.io.StreamCorruptedException
{
    Object k = maskNull(key);
    Object[] tab = table;
    int len = tab.length;
    int i = hash(k, len);

    Object item;
    while ( (item = tab[i]) != null) {
        if (item == k)
            throw new java.io.StreamCorruptedException();
        i = nextKeyIndex(i, len);
    }
    tab[i] = k;
    tab[i + 1] = value;
}
```

Serialization: readObject

Observation

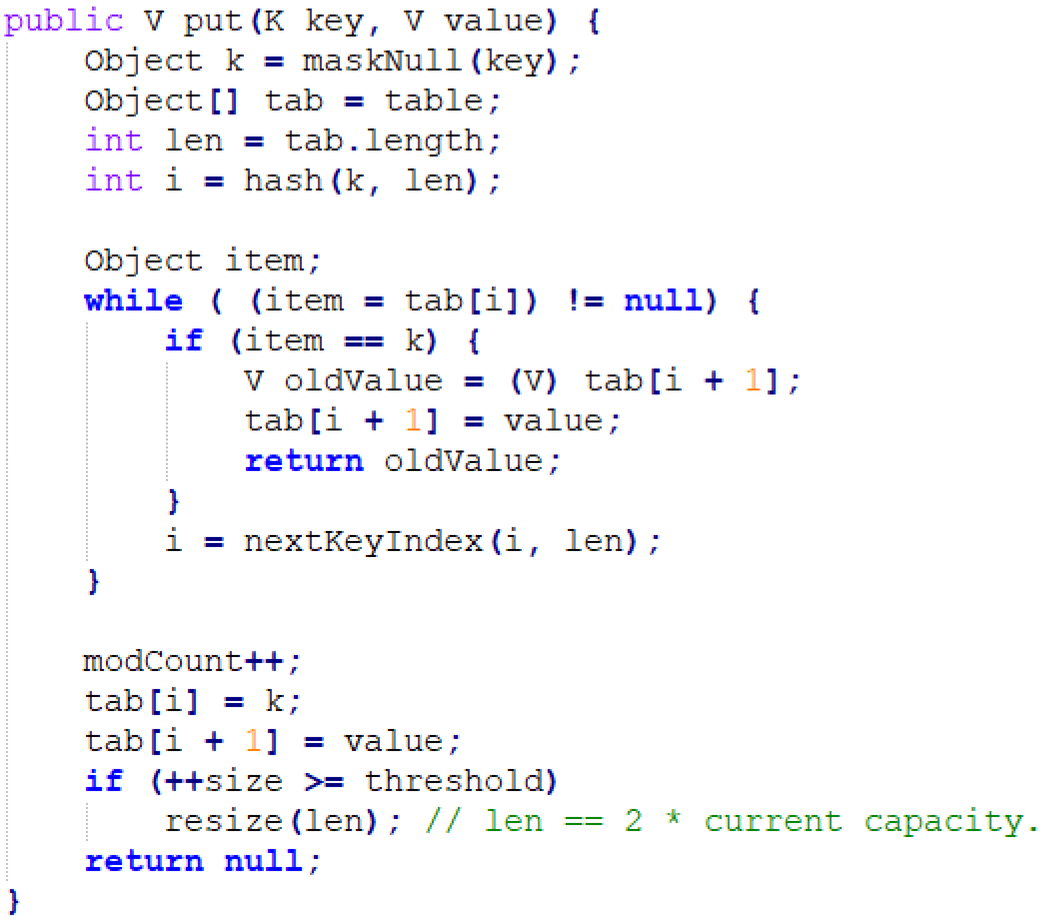
- Effectively, readObject is a constructor
- Constructors should establish class invariant

Potential security issue

- Attacker uses hex editor to modify file with hash map, say with size ≥ MAX_CAPACITY (and new entries)
- Victim deserializes the rogue hash map with readObject, which creates table array of MAX_CAPACITY
- Infinite loop triggered in putForCreate: no empty slot

Rough idea for fix: perform input validation to ensure that the stored IdentityHashMap satisfies the class invariant

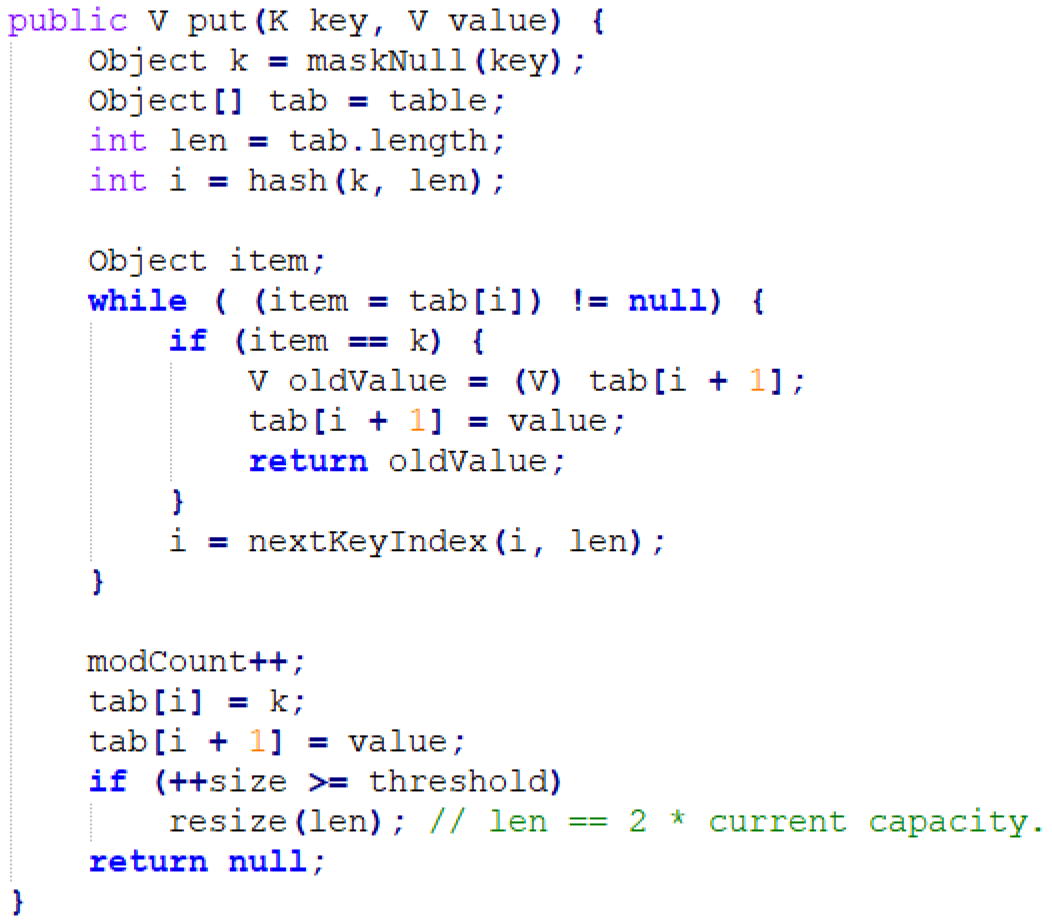
put



```
public V put(K key, V value) {
    Object k = maskNull(key);
    Object[] tab = table;
    int len = tab.length;
    int i = hash(k, len);

    Object item;
    while ( (item = tab[i]) != null) {
        if (item == k) {
            V oldValue = (V) tab[i + 1];
            tab[i + 1] = value;
            return oldValue;
        }
        i = nextKeyIndex(i, len);
    }

    modCount++;
    tab[i] = k;
    tab[i + 1] = value;
    if (++size >= threshold)
        resize(len); // len == 2 * current capacity.
    return null;
}
```

- adds a key/value to the table
- `resize`s (allocates new table array) if the load factor becomes larger than 2/3, except when the table is already at MAX_CAPACITY
- If size = MAX_CAPACITY-1 then resize throws an exception

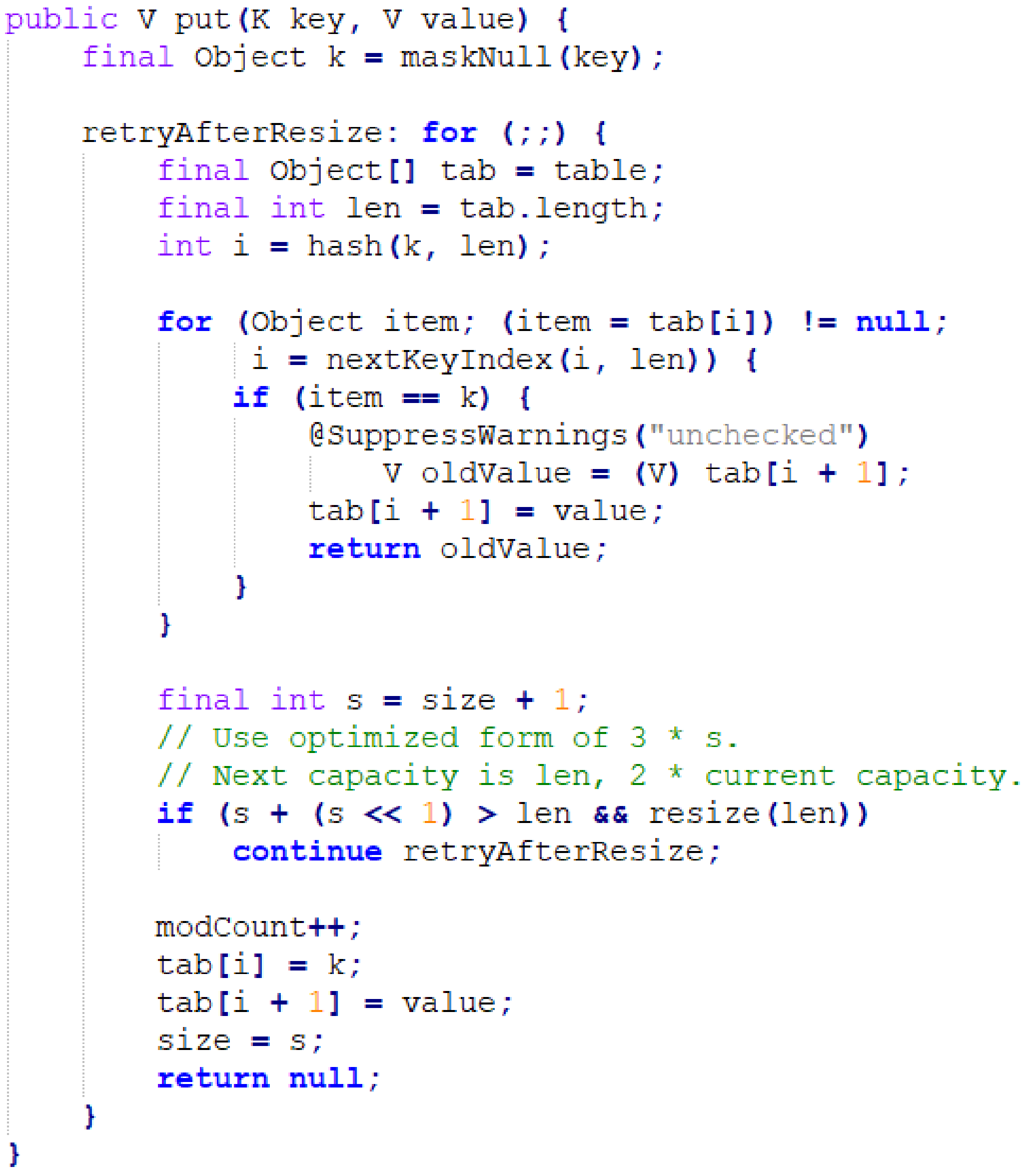
put



```
public V put(K key, V value) {
    Object k = maskNull(key);
    Object[] tab = table;
    int len = tab.length;
    int i = hash(k, len);

    Object item;
    while ( (item = tab[i]) != null) {
        if (item == k) {
            V oldValue = (V) tab[i + 1];
            tab[i + 1] = value;
            return oldValue;
        }
        i = nextKeyIndex(i, len);
    }

    modCount++;
    tab[i] = k;
    tab[i + 1] = value;
    if (++size >= threshold)
        resize(len); // len == 2 * current capacity.
    return null;
}
```

- Exception only thrown *after* table is modified!
- Modified table has no empty slot anymore: breaks class invariant
- No failure atomicity
- Map is corrupted, cannot be used afterwards in operations like get, containsKey, etc.: trigger infinite loop because no empty slot
- Vulnerability exploitable in DoS attack?

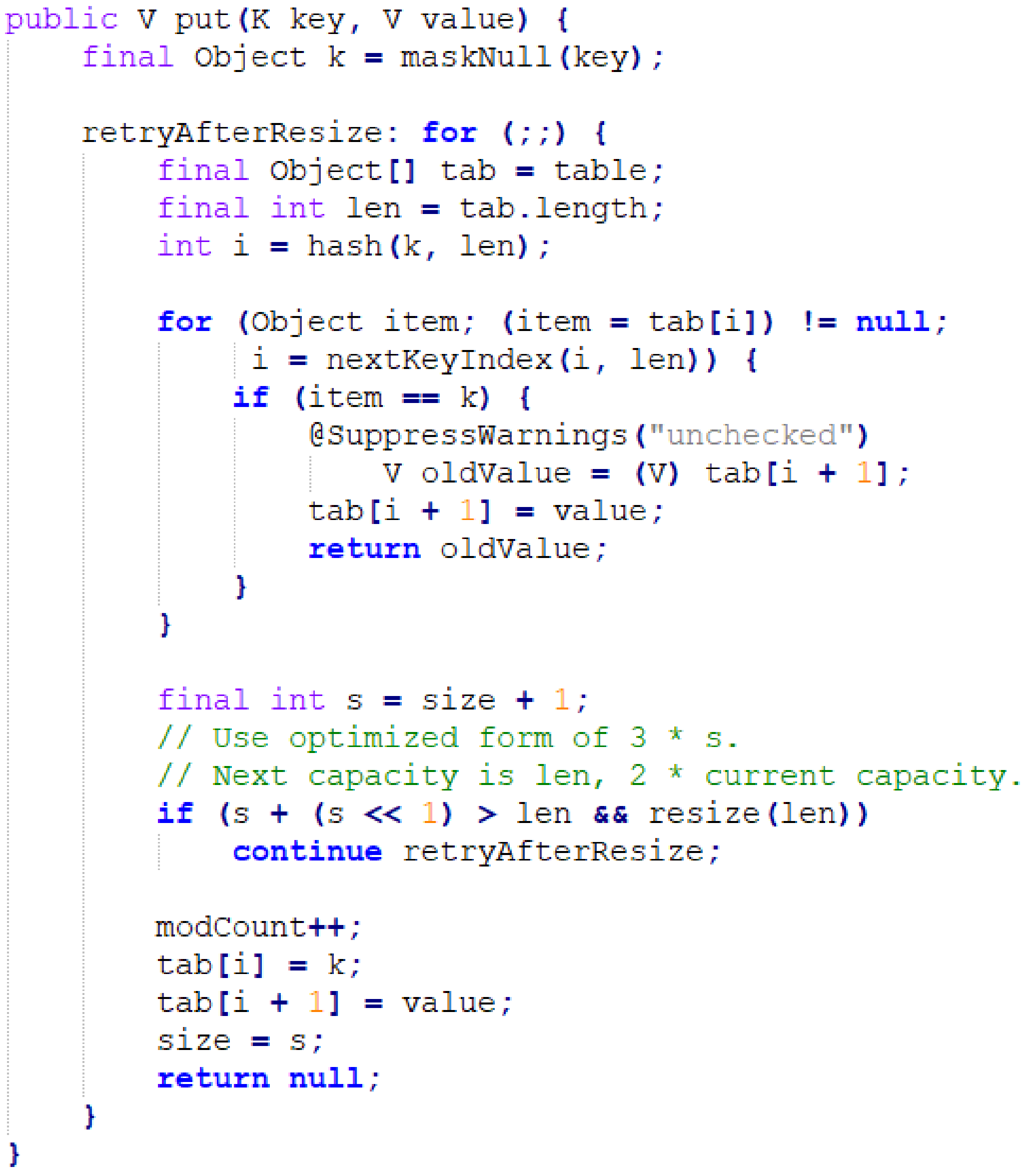
New put in later JDK update



```
public V put(K key, V value) {
    final Object k = maskNull(key);

    retryAfterResize: for (;;) {
        final Object[] tab = table;
        final int len = tab.length;
        int i = hash(k, len);

        for (Object item; (item = tab[i]) != null;
             i = nextKeyIndex(i, len)) {
            if (item == k) {
                @SuppressWarnings("unchecked")
                    V oldValue = (V) tab[i + 1];
                tab[i + 1] = value;
                return oldValue;
            }
        }

        final int s = size + 1;
        // Use optimized form of 3 * s.
        // Next capacity is len, 2 * current capacity.
        if (s + (s << 1) > len && resize(len))
            continue retryAfterResize;

        modCount++;
        tab[i] = k;
        tab[i + 1] = value;
        size = s;
        return null;
    }
}
```

- Map not modified if resize fails: **failure atomicity**
- resize rehashes all keys based on new table length, entries may move to very different index
- So: insertion point for key must be determined from scratch after resize
- Ugly control-flow to determine insertion point: empty for-loop just to use continue

put in newer JDK



```
public V put(K key, V value) {
    final Object k = maskNull(key);

    retryAfterResize: for (;;) {
        final Object[] tab = table;
        final int len = tab.length;
        int i = hash(k, len);

        for (Object item; (item = tab[i]) != null;
             i = nextKeyIndex(i, len)) {
            if (item == k) {
                @SuppressWarnings("unchecked")
                    V oldValue = (V) tab[i + 1];
                tab[i + 1] = value;
                return oldValue;
            }
        }

        final int s = size + 1;
        // Use optimized form of 3 * s.
        // Next capacity is len, 2 * current capacity.
        if (s + (s << 1) > len && resize(len))
            continue retryAfterResize;

        modCount++;
        tab[i] = k;
        tab[i + 1] = value;
        size = s;
        return null;
    }
}
```

- Better fix: extract method refactoring with loop that searches the index of a key, or its insertion point
    - return the index (positive number) if key is found
    - return negative index if key is not found (e.g. -10 if key should be inserted at index 10)

- Avoids code duplication and ugly control-flow: call the new helper method in put, get, etc.

Conclusion

- Hybrid analysis can be useful to speed up writing good specs
  - If the effort to load the case and to use the tool is reasonably small
  - JJBMC most successful, found semantic errors
- Ongoing experiment with proof scripts
  - Should be more robust than proof files (rules in proof files explicitly refer to the index of formulas in the sequent, so adding a new clause in a spec may shift the index of existing clauses and break proof loading)
  - Scripts ideally automatically generated from user interactions
- Introduce additional strategy macros?
  - Automatically simplify arithmetical ops that do not overflow
  - Better heap simplification (currently generates terms like self.a = self.a)
- Failure to ensure class inv in deserialization may be widespread
- More details in iFM 2022 paper (paper title same as this talk).

Questions?