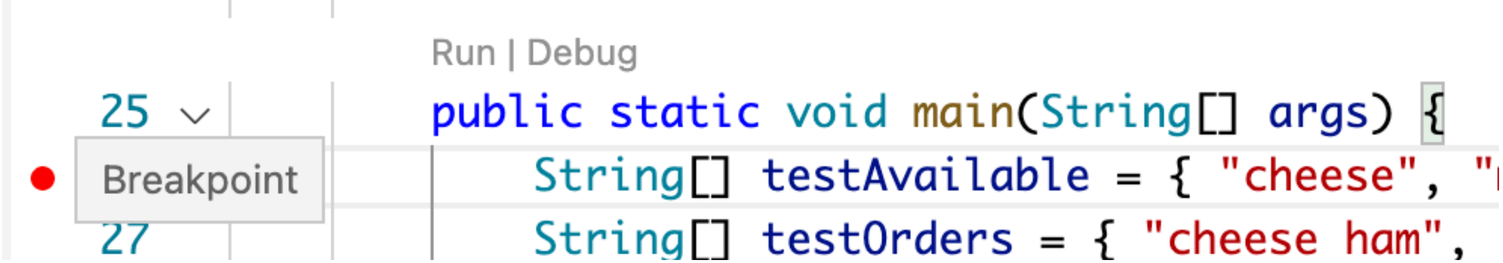
CompSci 201, L6: Hashing, HashMap, HashSet

Announcements, Coming up

- Today, Wednesday, 2/1
    - APT 2 due
- Monday, 2/6
    - Project 1: NBody due (future projects will be 2 week)
- Next Wednesday, 2/8
    - APT 3 due

Set a breakpoint



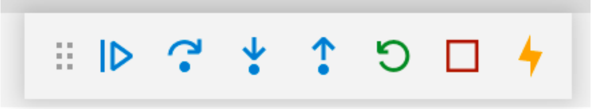
```
public static void main(String[] args) {
    String[] testAvailable = { "cheese", "
    String[] testOrders = { "cheese ham",
```

- Start by setting a *breakpoint* in your code.
- Says “run the program until the first time this line executes, then pause to step line by line.”
- If you want to go line by line from the beginning? Set to first line in main.

Debug options

See the documentation for the tool

Will see a menu like this:

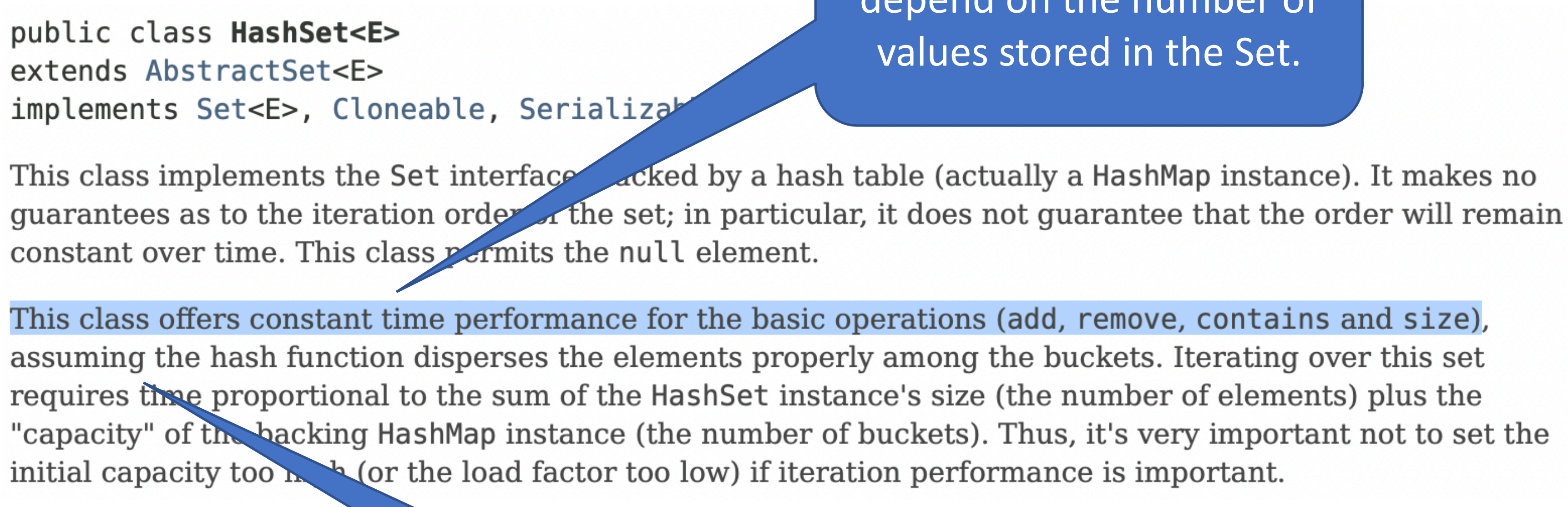


- Continue: Go to next breakpoint
- **Step over**: Execute line, go to next. Run whole methods.
- **Step into**: Same as over *unless method call*. Steps into methods, jumping to first line of method code.
- Step out: Break out of method back to where called
- Restart: Start over again at first breakpoint
- Stop: Stop debugging session

Live Debugger Demo

- Live coding

HashSet/Map efficiency



```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Constant time = does not depend on the number of values stored in the Set.

Under assumptions…

Java API documentation

Aside: Does constant time lookup (`contains()`, `get()`, etc.) matter?

- Social media: When you login, server needs to lookup to display the correct page for you.
  - Billions of accounts! Look it up in a List? NO! Constant time lookup with hashing.

- Routing/directions application: Need to lookup roads from a given intersection.
  - How many possible roads? Search through a list? NO! Constant time lookup with hashing.

- Could go on!

Big questions about hashing

Last class: Usage of API HashSet/HashMap.

Today:

1. How does a hash table work to implement HashMap/HashSet?
2. Why do `.equals()` and `.hashCode()` matter?
3. Why are the `add()`, `contains()`, `put()`, `get()`, and `containsKey()`, etc., all constant time (and under what assumptions)?

Hash Table Concept

- Implement HashMap with an Array also
  - Of <key, value> pairs
- Rather than adding to position 0, 1, 2, …
- **Big idea:** Calculate **hash** (an int) of key to determine where to store & lookup

hash("ok")== 4

0	
1	<"hi", 5>
2	
3	
4	<"ok", 3>
5	
6	
7	

  - Java OOP: Will use the hashCode() method of the key to get the hash
- Same hash to put and get, no looping over list

HashMap methods at a high level

Always start by getting the **hash** =
`Math.abs(key.hashCode()) % list.size()`

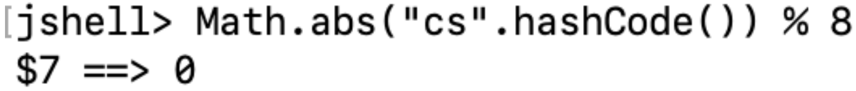
Absolute value and % (remainder when dividing by) list size ensures valid index

- put(key, value)
  - Add (<key, value>) to list at index hash
  - If key already there, update value
- get(key)
  - Return value paired with key at index hash position of list
- containsKey(key)
  - Check if key exists at index hash position of list

0	
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

HashMap put/get example

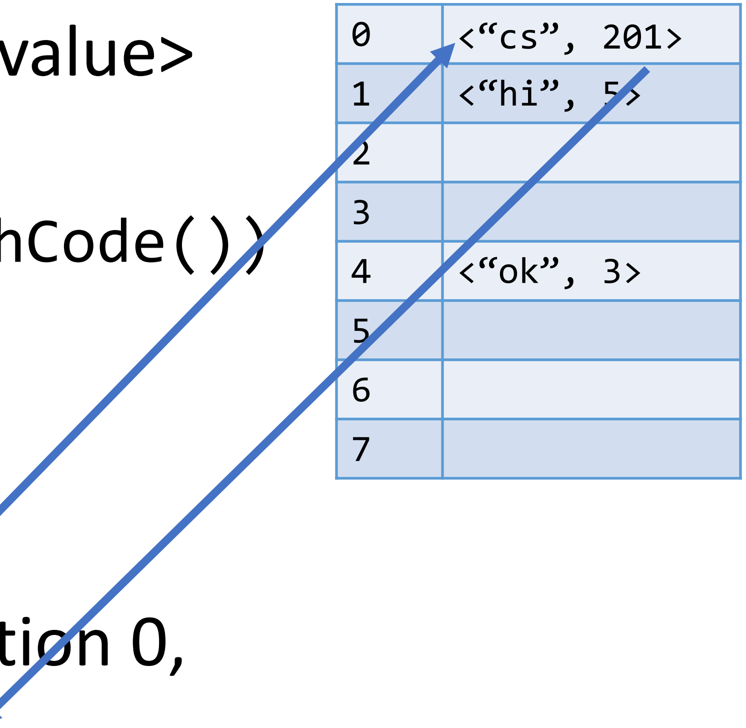
- Suppose we have the <key, value> pair <“cs”, 201>.
- hash is Math.abs(“cs”.hashCode()) % 8 which is 0.



```
jshell> Math.abs("cs".hashCode()) % 8
$7 ==> 0
```

- put(“cs”, 201) in position 0
- get(“cs”) by looking up position 0, returning the value

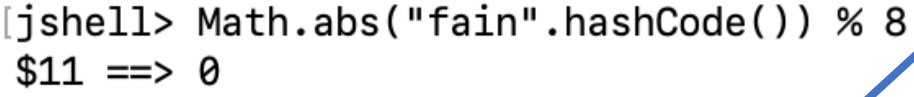
return 201



0	<“cs”, 201>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

Collisions

- Suppose now we want to put `<“fain”, 104>`.
- `hash=Math.abs(“fain”.hashCode()) % 8` which is `0`.



```
[jshell> Math.abs("fain".hashCode()) % 8
$11 ==> 0
```

- put(“fain”, 104) in position 0
- But `<“cs”, 201>` is already stored at position 0! Call this a **collision**.

0	<“cs”, 201>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

Dealing with collisions: concepts

- Think of the hash table as an Array of "buckets".
- Each bucket can store multiple <key, value> pairs.
- `put(key, value)`
  - Add to hash index bucket
  - Update value if key already in bucket
- `get(key)`
  - Loop over keys in hash index bucket
  - Return value of one that equals() key

0	<"cs", 201> <"fain", 104>
1	<"hi", 5>
2	
3	
4	<"ok", 3>
5	
6	
7	

Dealing with collisions: details

- Bucket is really another list.
- Hash table is really an array of of lists of <key, value> pairs.
- We call this technique for dealing with collisions **chaining**.

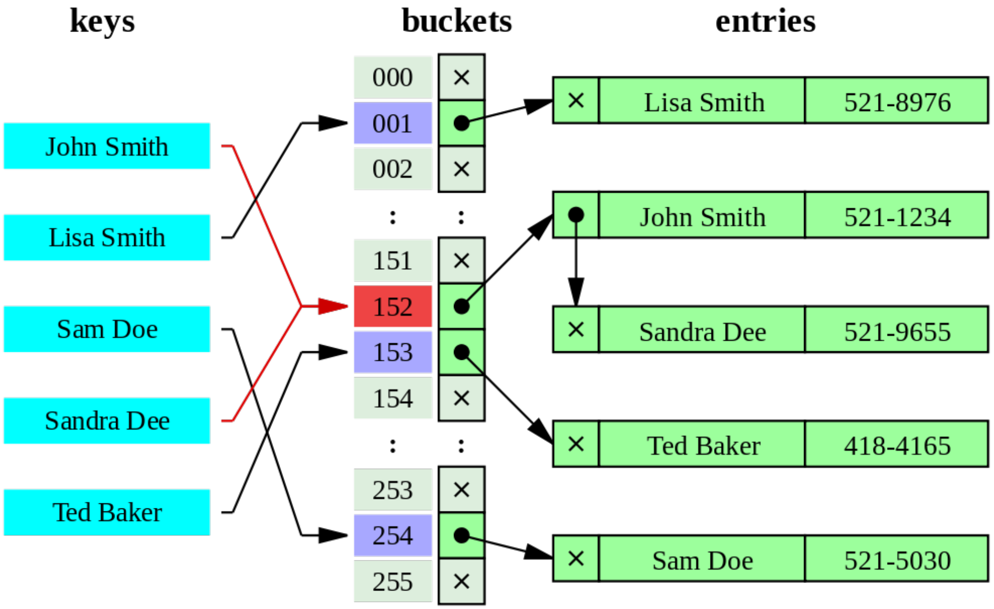


Illustration credit: By Jorge Stolfi - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6471915

WOTO

Go to duke.is/mxnt5

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

L06-WOTO1-Hash

* Required

* This form will record your name, please fill your name.

1. NetID *

2. HashSet and HashMap have constant time add, contains, put, get, and containsKey operations. That means that these methods... *

○ Take the same amount of time to run

○ Have the same number of operations

○ [✓ Runtimes do not depend on number of elements of the Set/Map]

3. What is stored in each "bucket" in a hash table / HashMap? *

○ A <key, value> pair

○ A list of keys

○ A list of values

○ [✓ A list of <key, value> pairs]

4. Suppose we want to put <s, 1> into a HashMap where s.hashCode() = 12. If our hash table has 4 buckets, in which bucket will we store <s, 1>? *

☑ 0

○ 1

○ 2

○ 3

○ 4

○ 5

○ 6

○ 7

○ 8

○ 9

○ 10

○ 11

○ 12

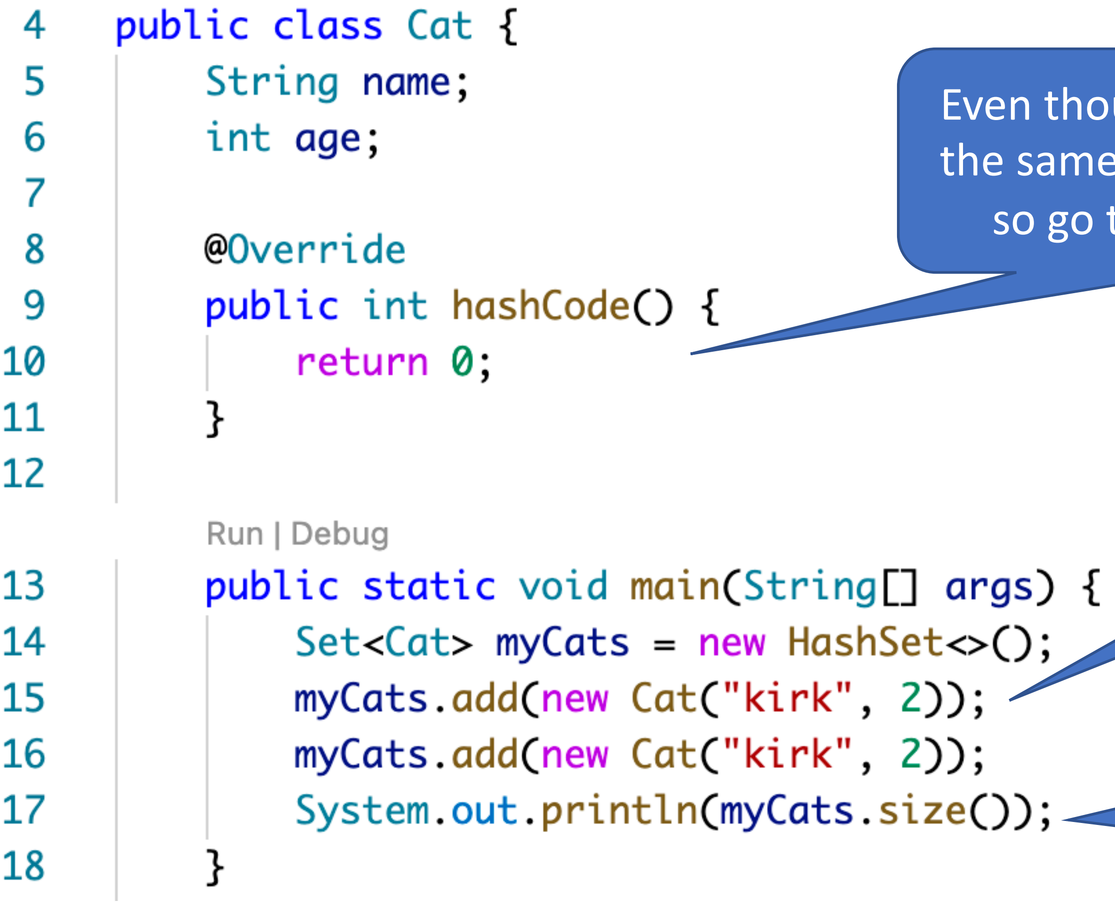
Where does `equals()` come in?

- If multiple <key, value> pairs in same bucket, need to know which to get() or update on a put() call.

- Always the pair where the key in the bucket `equals()` the key we `put()` or `get()`.

- Need `equals()` to work correctly for the key type
  - String keys? Integer? Already implemented for you.
  - Storing objects of a class *you write*? Need to override and implement `equals()`.

What happens without equals()? Hashing cats



```java
public class Cat {
    String name;
    int age;

    @Override
    public int hashCode() {
        return 0;
    }

    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats.size());
    }
```

Even though all cat objects have the same hashCode( ) of 0 and so go to the same bucket…

And these 2 Cat objects have the same values

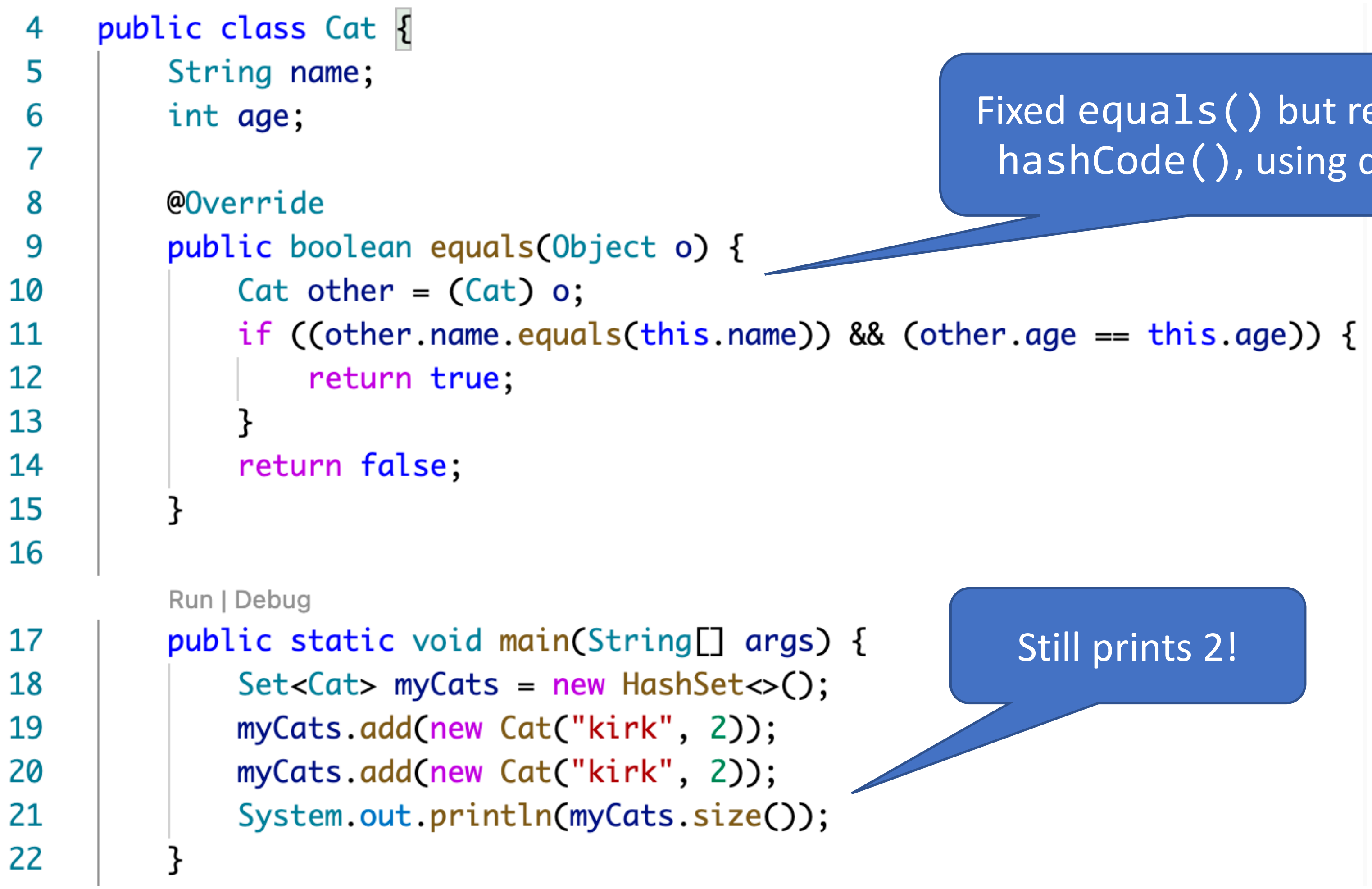
Prints 2, cannot detect duplicates without equals( )

hashCode Correctness

- Need `hashCode()` to work correctly for the key type.
  - String keys? Already implemented for you.
  - Storing objects of classes *you write*? Need to override and implement `hashCode()`.

- What makes a `hashCode()` "correct" (not necessarily efficient)?
  - **Any two objects that are `equals()` should have the same `hashCode()`.**

What happens without hashCode()? Hashing more cats

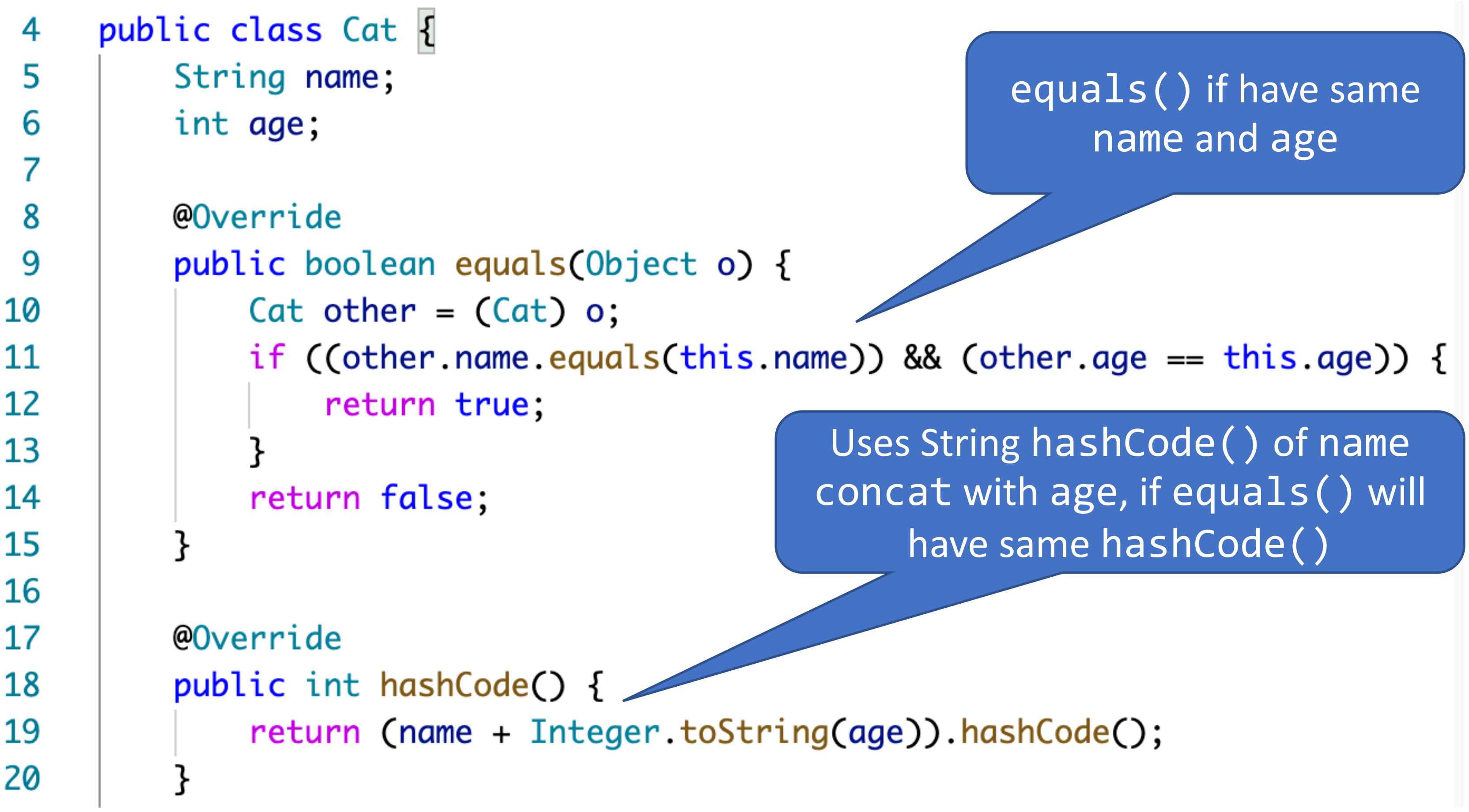


```
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name.equals(this.name)) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats.size());
    }
```

Fixed equals() but removed hashCode(), using default

Still prints 2!

Cat with `equals()` and `hashCode()`



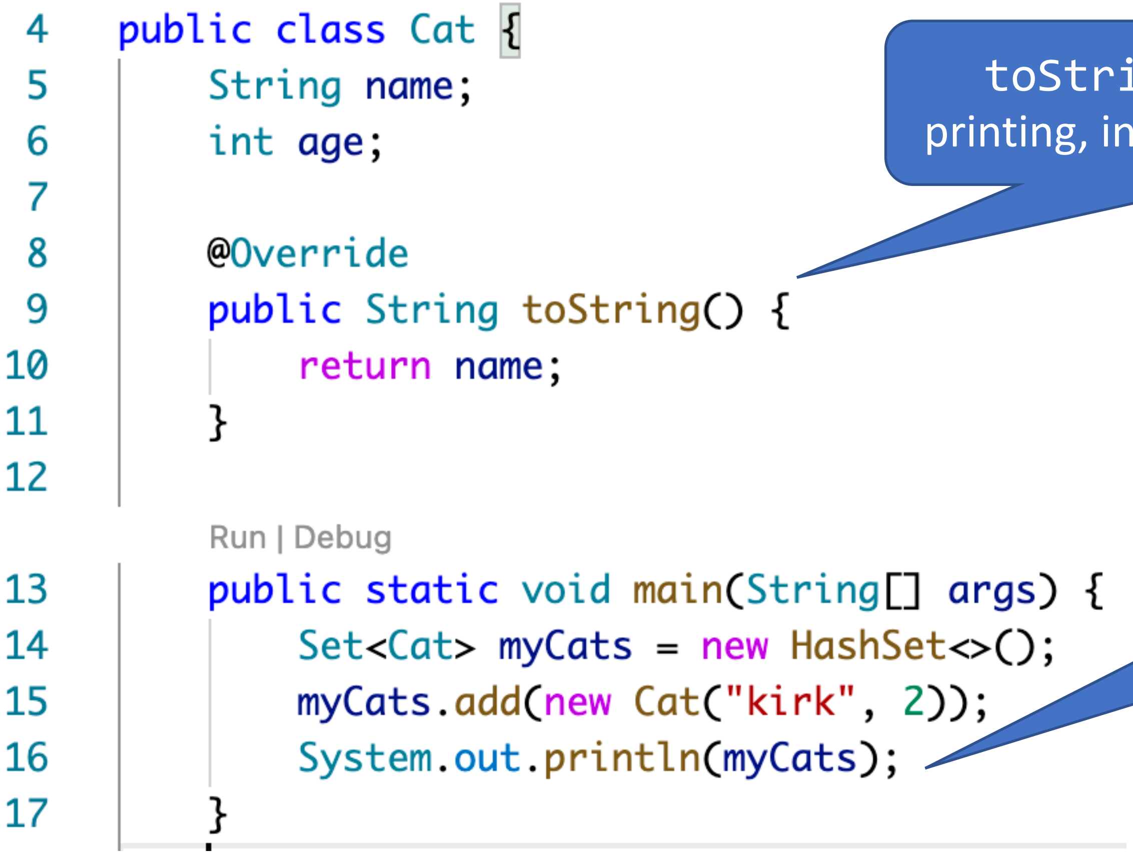
```java
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name.equals(this.name)) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
```

`equals()` if have same name and age

Uses String `hashCode()` of name concat with age, if `equals()` will have same `hashCode()`

Aside: toString()

Don't need for hashing, but toString() method allows "nice" printing.



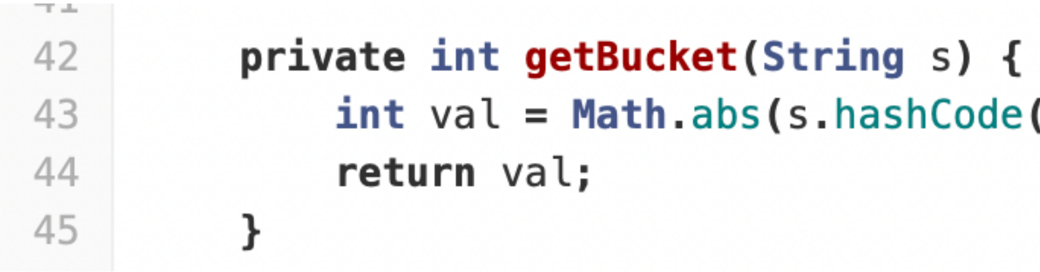
```java
public class Cat {
    String name;
    int age;

    @Override
    public String toString() {
        return name;
    }

    public static void main(String[] args) {
        Set<Cat> myCats = new HashSet<>();
        myCats.add(new Cat("kirk", 2));
        System.out.println(myCats);
    }
```

toString() method used for printing, including inside a Collection

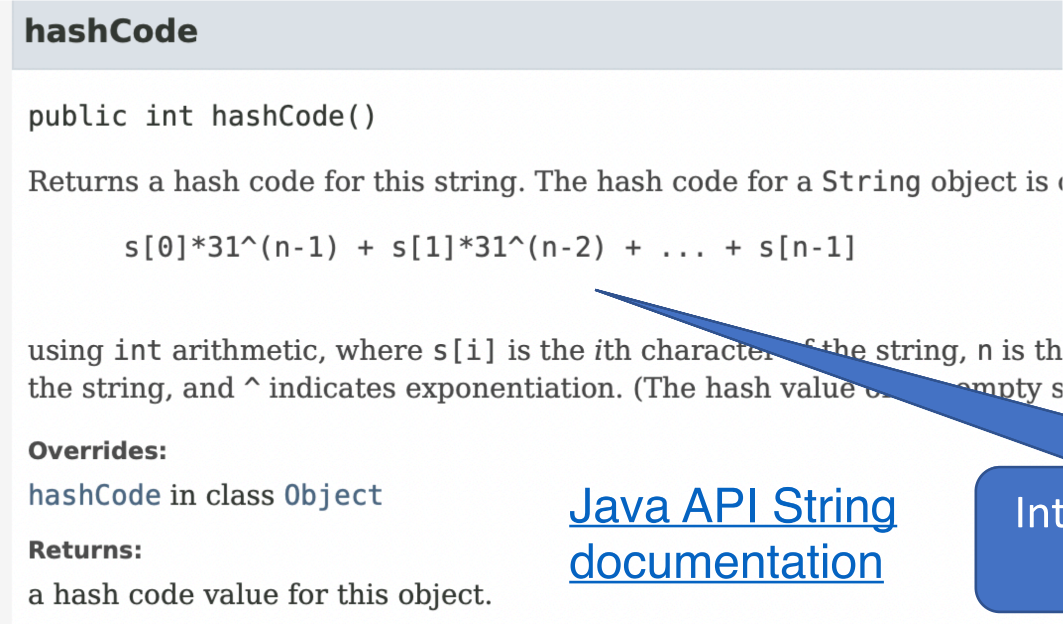
Prints [kirk] instead of [Cat@...]

What is the String hashCode()?

Remember how hashCode() is used to get the bucket index.



```
    private int getBucket(String s) {
        int val = Math.abs(s.hashCode()) % myTable.size();
        return val;
    }
```



hashCode

`public int hashCode()`

Returns a hash code for this string. The hash code for a `String` object is c

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

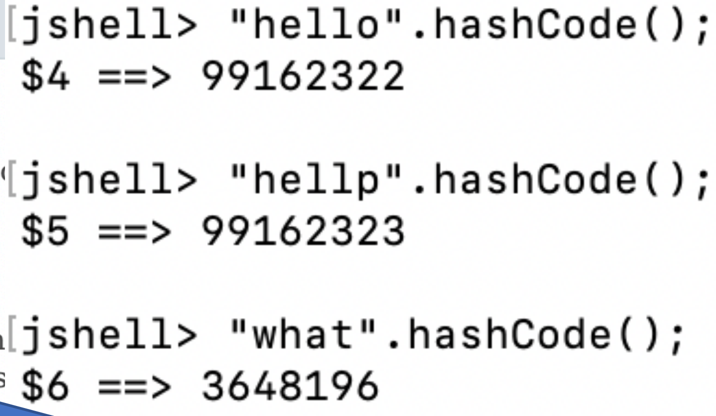
using `int` arithmetic, where `s[i]` is the *i*th characte f the string, `n` is th
the string, and `^` indicates exponentiation. (The hash value o empty s

Overrides:

`hashCode` in class `Object`

Returns:

a hash code value for this object.



```
jshell> "hello".hashCode();
$4 ==> 99162322

jshell> "hellp".hashCode();
$5 ==> 99162323

jshell> "what".hashCode();
$6 ==> 3648196
```

[Java API String documentation](#)

Interprets each character as an int, does arithmetic.

Revisiting Hashing Efficiency

- Real runtime of `get()`, `put()`, and `containsKey() =`
  - Time to get the hash
  - + Time to search over the hash index "bucket", calling .equals() on everything in the bucket

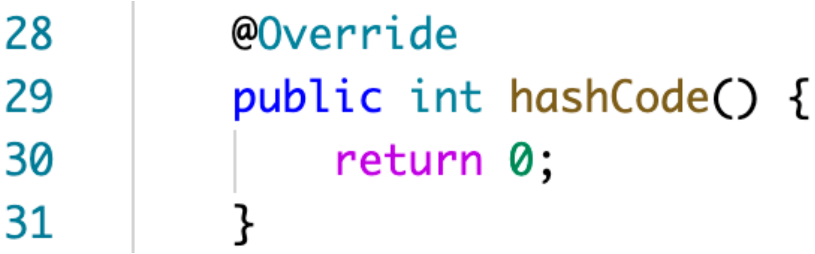
Constant, does not depend on number of pairs in Map

Depends on number of pairs per bucket

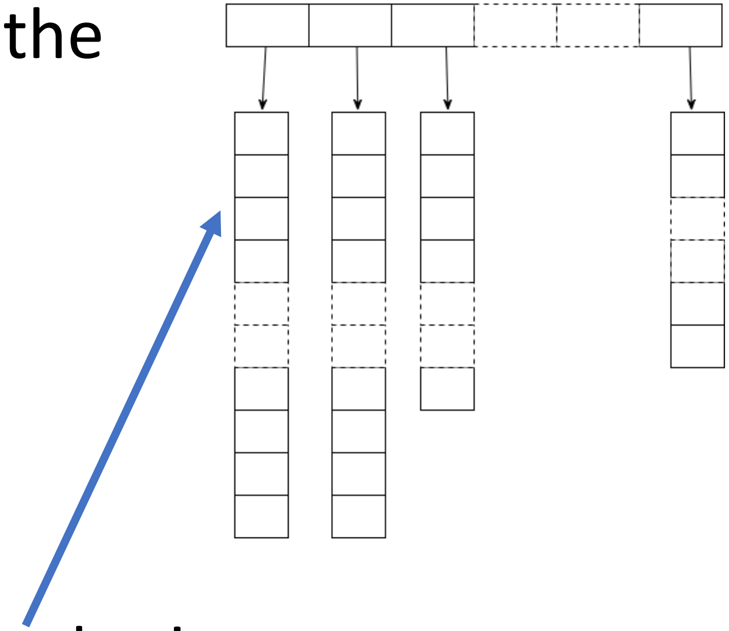
→HashMaps faster with more buckets

"correct" but inefficient hashCode()

Correctness requirement: Any .equals() keys should have the same hashCode().



```
@Override
public int hashCode() {
    return 0;
}
```



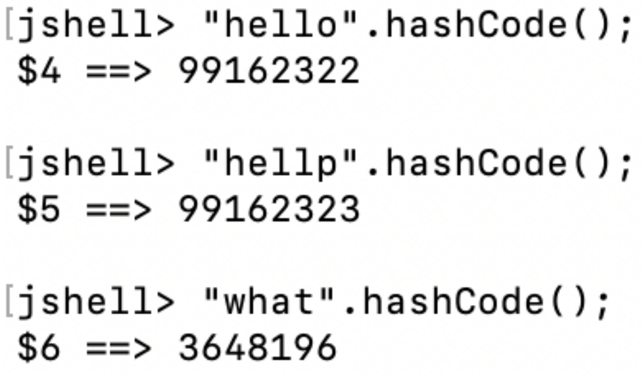
Still satisfies, but not good…

Stores everything in the first bucket!
No more efficient than a list!

Correct and efficient `hashCode()`

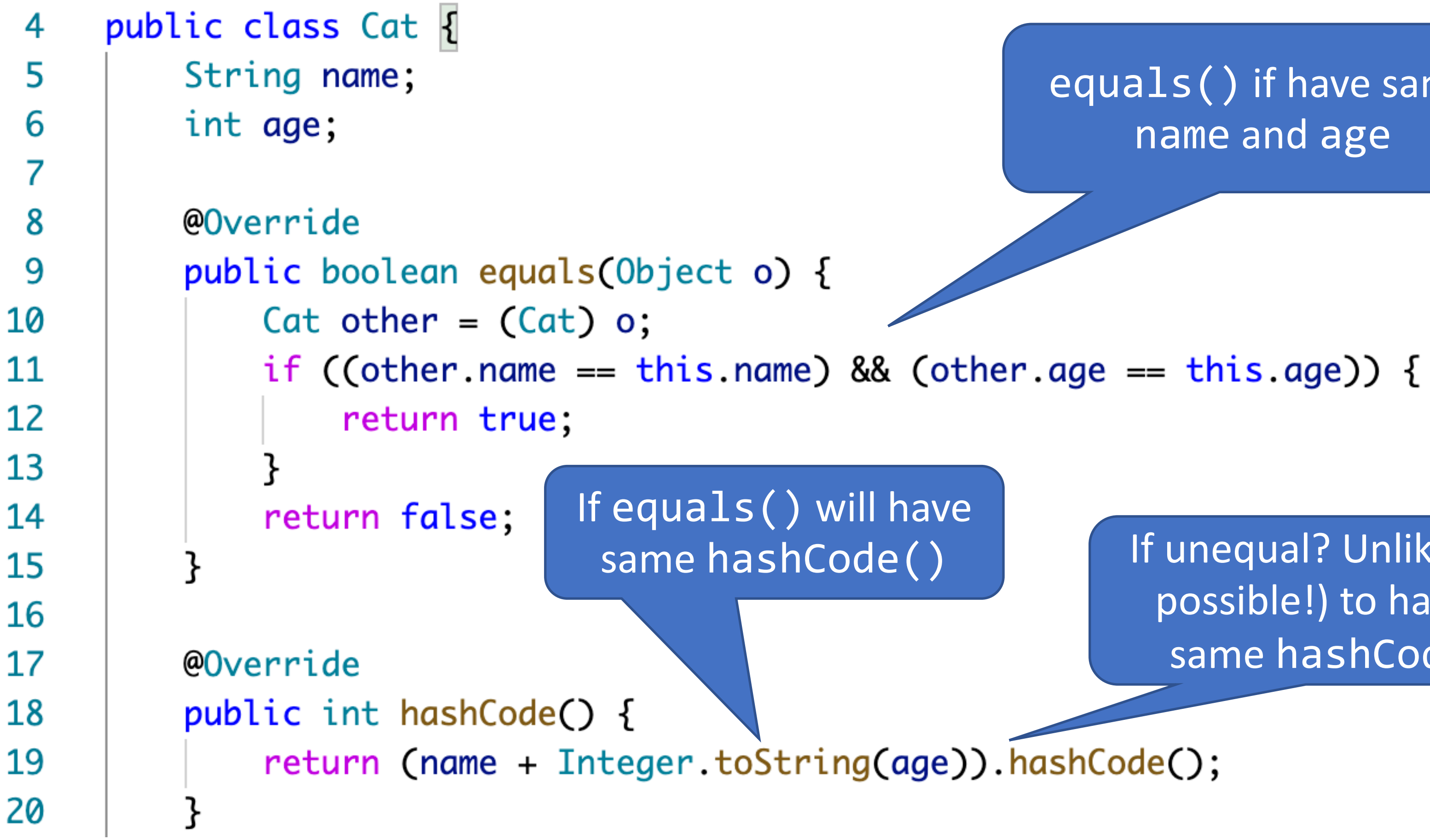
From the Java 17 API documentation:

- Correctness: “If two objects are equal…hashCode…must produce the same integer result.”
- Efficiency: “…producing distinct integer results for unequal objects may improve the performance of hash tables.”
- String hashCode() satisfies both



```
[jshell> "hello".hashCode();
 $4 ==> 99162322

[jshell> "hellp".hashCode();
 $5 ==> 99162323

[jshell> "what".hashCode();
 $6 ==> 3648196
```

Cat hashCode() revisited



```
public class Cat {
    String name;
    int age;

    @Override
    public boolean equals(Object o) {
        Cat other = (Cat) o;
        if ((other.name == this.name) && (other.age == this.age)) {
            return true;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (name + Integer.toString(age)).hashCode();
    }
```

equals() if have same name and age

If equals() will have same hashCode()

If unequal? Unlikely (but possible!) to have the same hashCode().

Simple uniform hashing assumption (SUHA)

- Suppose we hash N pairs to M buckets.

- **Simple uniform hashing assumption:** Probability two random (unequal) keys hash to same bucket is just 1/M.
  - Spread of pairs to buckets **looks random** (but is not).
  - Ways to design such hash functions, not today
  - We will make the assumption to analyze efficiency in theory, can verify runtime performance in practice

Implications of SUHA

- Expected number of pairs per bucket under SUHA? N/M [N pairs, M buckets].
- Stronger statements are true: Very high probability that a bucket has approximately N/M pairs.
- Runtime implication?
  - Time to get the hash
  - Time to search over the hash index “bucket”
    - Calling .equals() on everything in the bucket

Constant, does not depend on N or M.

Expect ~ N/M pairs to search

Memory/Runtime Tradeoff

- N pairs, M buckets, assuming SUHA / good hashCode()

- **Case 1: N >> M** – too many pairs in too few buckets
  - Runtime inefficient
- **Case 2: M >> N** – too many buckets, not many pairs
  - Runtime efficient, NOT memory efficient
- **Case 3: M slightly larger than N** – sweet spot
  - Runtime efficient, memory usage slightly more than an array/ArrayList

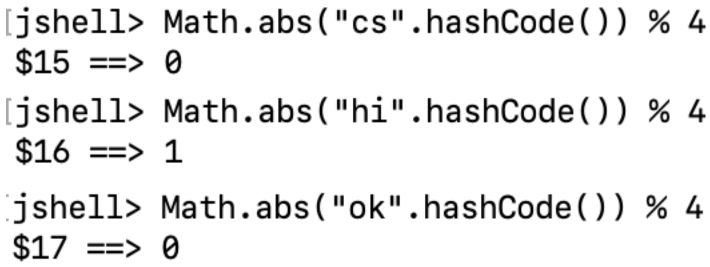
Load Factor and HashMap Growth

- N pairs, M buckets

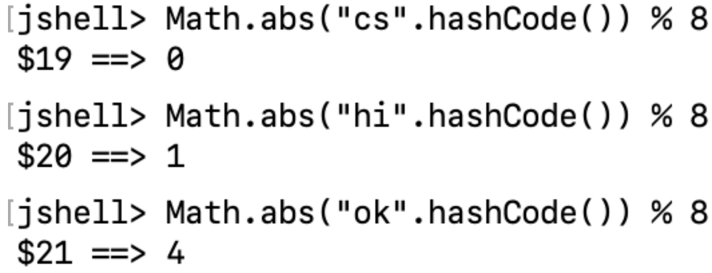
- Load factor = maximum N/M ratio allowed
  - Java default is 0.75

- Whenever N/M exceeds the load factor?
  - Create a new larger table, rehash/copy everything
  - Double the size, geometric growth pattern for amortized efficiency just like ArrayList!
  - Called resizing

Hash table resizing



```
jshell> Math.abs("cs".hashCode()) % 4
$15 ==> 0
jshell> Math.abs("hi".hashCode()) % 4
$16 ==> 1
jshell> Math.abs("ok".hashCode()) % 4
$17 ==> 0
```



```
jshell> Math.abs("cs".hashCode()) % 8
$19 ==> 0
jshell> Math.abs("hi".hashCode()) % 8
$20 ==> 1
jshell> Math.abs("ok".hashCode()) % 8
$21 ==> 4
```

0	<“cs”, 201> <“ok”, 3>
1	<“hi”, 5>
2	
3	

Resizing

0	<“cs”, 201>
1	<“hi”, 5>
2	
3	
4	<“ok”, 3>
5	
6	
7	

WOTO

Go to duke.is/8khxt

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

L06-WOTO2-Hashing

* Required

* This form will record your name, please fill your name.

1. NetID *

2. Which methods must be correctly implemented in order for a HashSet/HashMap to function correctly? Select all that apply. *

☑ equals() for the key objects

□ equals() for the value objects

☑ hashCode() for the key objects

□ hashCode() for the value objects

3. Suppose you store one million (1,000,000) Keys in a HashSet where the hashCode() of all the keys returns 0 but none of the keys are equal to each other (according to equals()). What would you expect when calling contains() on the HashSet? *

○ Incorrect behavior, returning the wrong value

○ Correct and efficient behavior, constant time

☑ Correct and inefficient behavior, comparable to contains in ArrayList

○ None of the above

4. Suppose a HashSet/Map performs a resizing operation to double the number of buckets every time it reaches a load factor of 1. Assume a good implementation of hashCode() for the keys / the simple uniform hashing assumption. When performing N add/put operations with unique keys, the best characterization of the runtime complexity of add/put is... *

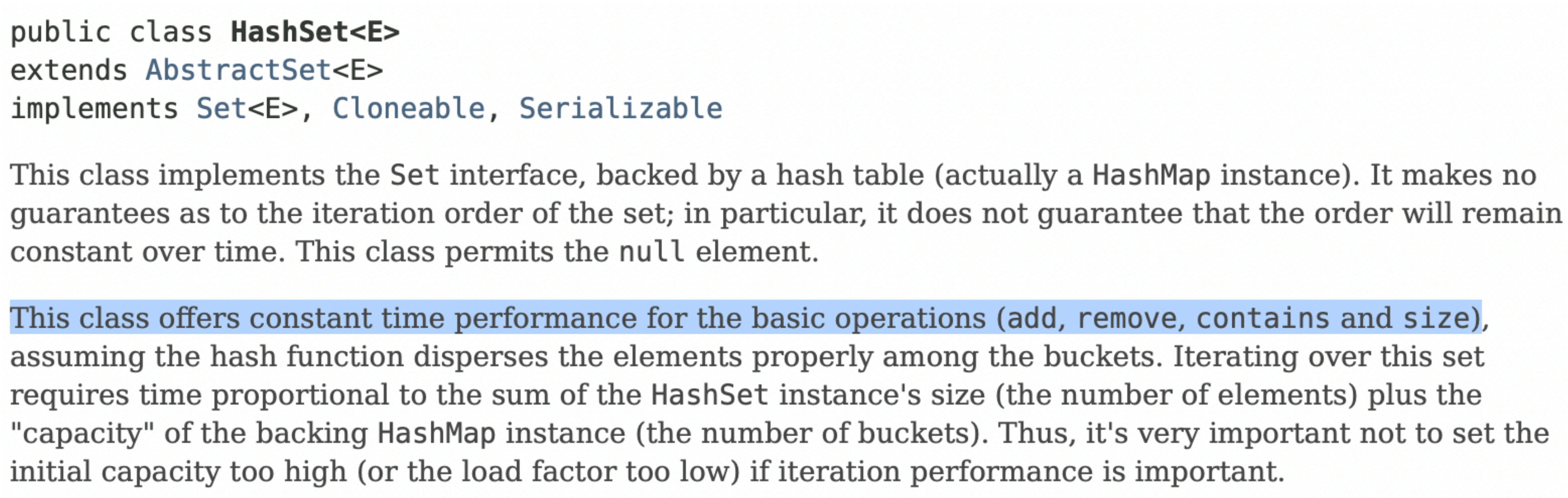
○ Constant time

○ Amortized constant time

○ Expected constant time

☑ Amortized expected constant time

Revisiting guarantees



```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the `null` element.

This class offers constant time performance for the basic operations (`add`, `remove`, `contains` and `size`), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the `HashSet` instance's size (the number of elements) plus the "capacity" of the backing `HashMap` instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Java API documentation

Constant *amortized* time operations *in expectation* under the simple uniform hashing assumption (practically, assuming the hash function distributes unequal keys).