Lecture 7 – Lists & Maps

08-671
Java Programming for App Developers

September 22, 2015

Jeffrey L. Eppinger & Terry Lee

08-671 Lecture Topics

(subject to change – but only a little bit)

#1 Intro
#2 Primitive Types
#3 Java Classes
#4 Reference Types
#5 Loops & Arrays
#6 Methods & Classes
#7 Lists & Maps
#8 File & Network I/O
#9 Swing Interfaces
#10 Swing Actions
#11 Threads
#12 Exceptions
#13 Functional Programming
#14 In-class Written Exam

* Final Exam – this will be a 3-hour programming problem

Outline

⇢Questions

Lists

Interfaces& Generics

Maps

Sorting, again

Autoboxing

HW#5

Question for You

- What's the hardest part about writing checks?

Example

- AlphaNumber.java

What does "null" mean?

- For reference variables may not be pointing at anything
  - The value is "null"

```
        String s = null;
        while (s != null) { … }
```

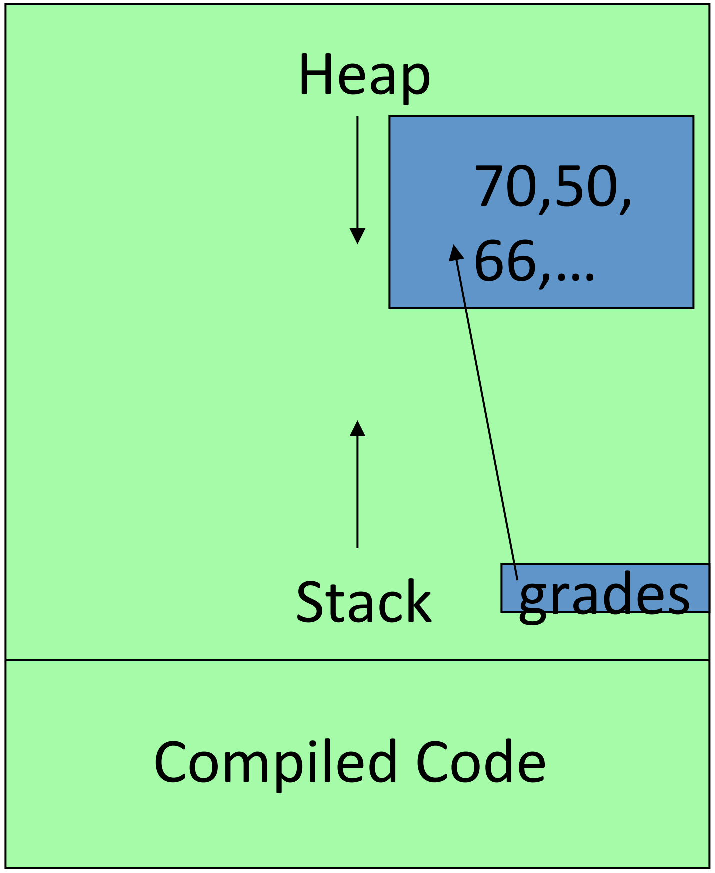
Outline

- ✓ Questions
- ⇢ Lists
- Interfaces & Generics
- Maps
- Sorting, again
- Autoboxing
- HW#5

What's a List?

- Examples:
  - Shopping List
  - To Do List
- How do we store the list?
  - An array?
  - Example: StringArray.java

Remember Arrays?



Remember the Hassles

1. Knowing how much storage to allocate

```
        String a[] = new String[10];
```

2. Appending:

```
        String item = …;
        a[count] = item;  count = count + 1;
```

3. Inserting before the first element:

```
        for (int i=count; i>0; i--) a[i] = a[i-1];
        a[0] = item;  count = count + 1;
```

4. For each:

```
         for (String s : a) if (s …
         for (int i=0; i<count; i++) if (a[i] …
```

5. Deleting, 6. Searching, …

java.util.ArrayList

- Implements an expandable array
  - Internally maintains an array
  - Provided methods to access the array
  - When the array overflows, it allocates a new larger array and copies the data from old (smaller) array into the new (larger) array

Just use ArrayList!

Automatically handles issues relating to size

0. Import `java.util.ArrayList;`
1. Allocation of space

```
        List<String> a;
        a = new ArrayList<String>();
```

2. Appending

```
        String item = …;
        a.add(item);
```

3. Inserting before the first element

```
        a.add(0,item);
```

ArrayList -- continued

4. For each

```
for (String s : a) if (s …
for (int i=0; i<a.size(); i++)
    if (a.get(i)…
```

5. Deleting

```
a.remove(i);
```

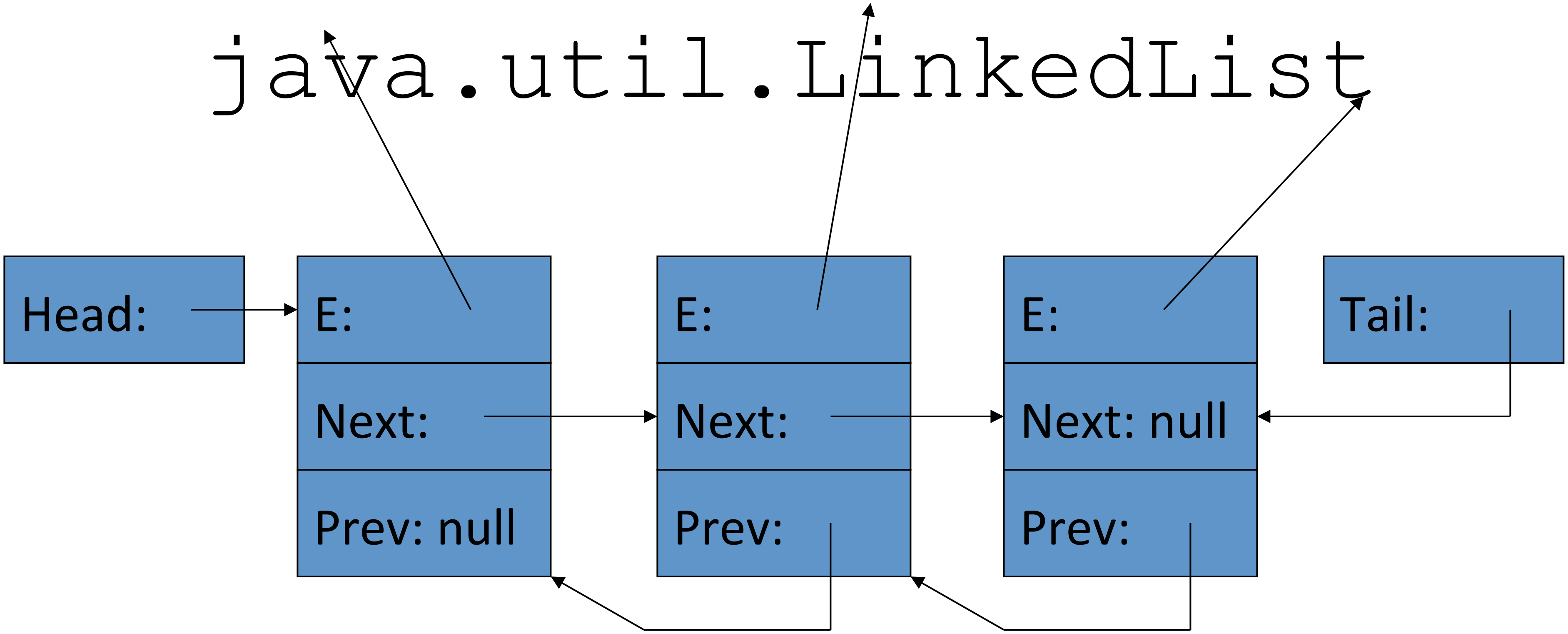
6. Searching

```
i = a.indexOf(item);
```

ArrayList Manages an Array (`E[]`)

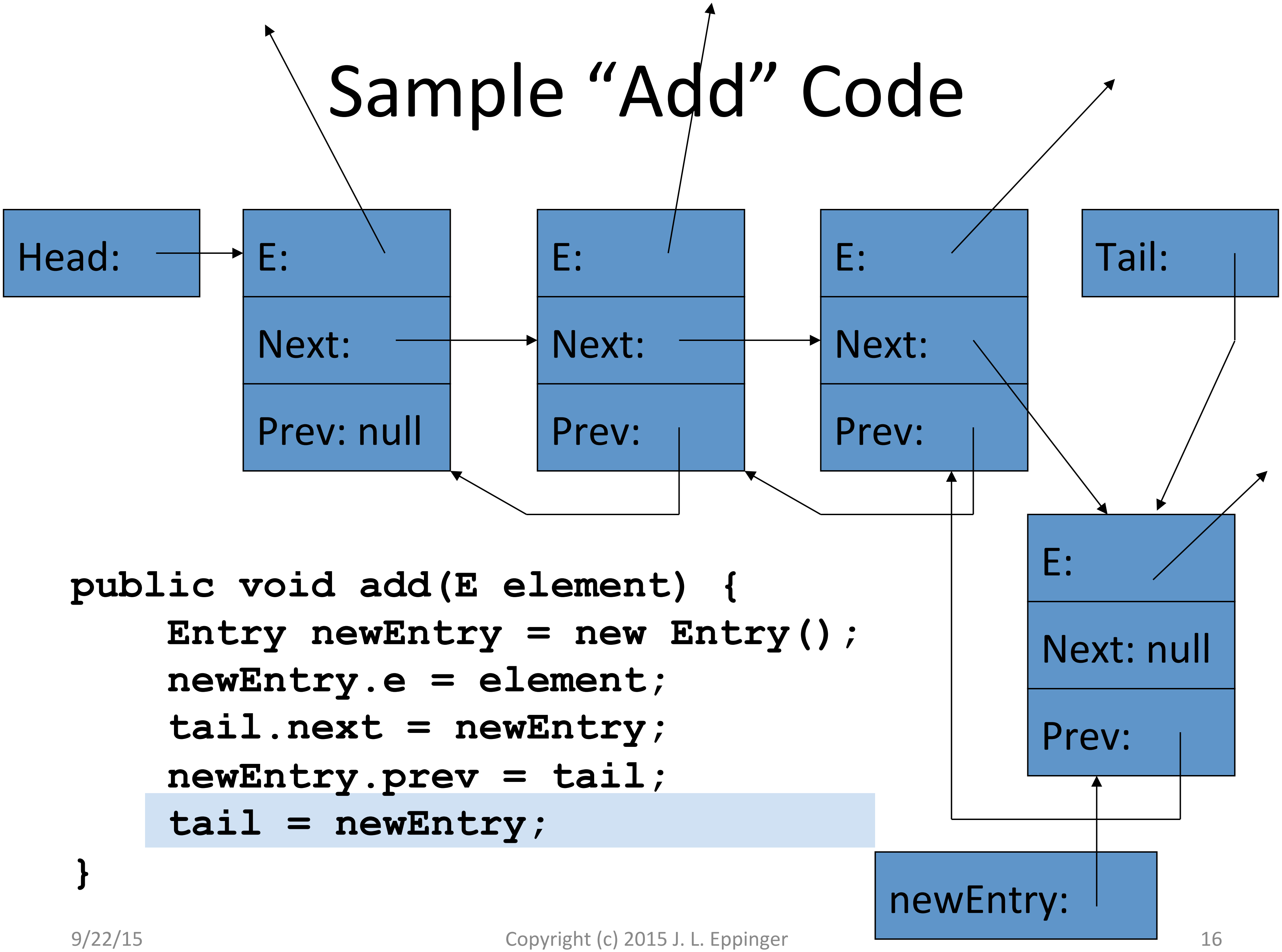
- From the point of view of the user of an ArrayList:
  - The array has initial *size* of 0
  - Can be grown or shrunk by adding or removing elements
- Implementation
  - Makes a internal array of size 10 – this is the *capacity*
  - Maintains a separate count of the number of elements <E> referenced by the array – the user's perception of *size*
  - If adding an element exceeds the *capacity*, allocate a new array that is 50% bigger and copy of data from the old array into (the beginning of) the new array

java.util.LinkedList



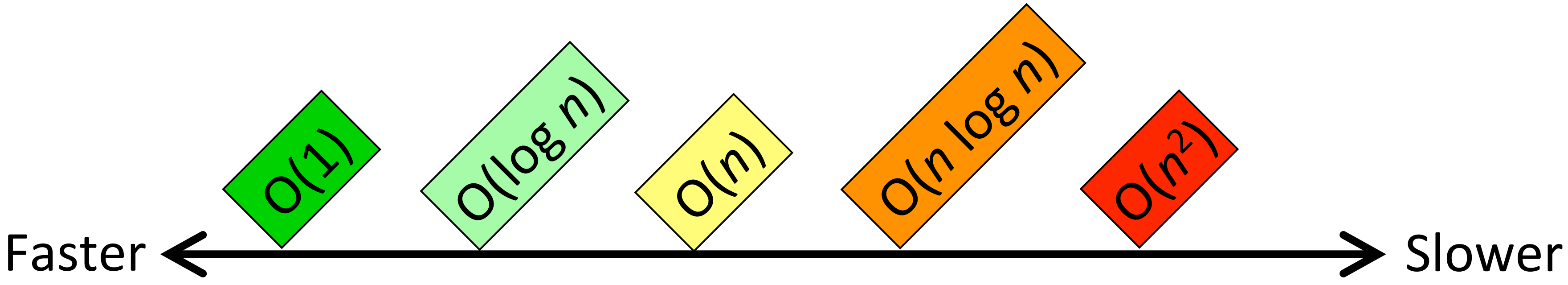
- Elements <E> are referenced by Entry records
- To add an element, a new Entry record is allocated and inserting at the beginning, end or middle of the list

Sample “Add” Code



```
public void add(E element) {
    Entry newEntry = new Entry();
    newEntry.e = element;
    tail.next = newEntry;
    newEntry.prev = tail;
    tail = newEntry;
}
```

Fast and Slow



Faster

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

Slower

n is the number of elements

c is some constant

O(1) means an operation take time less than c

O(log n) means an operation take time less than c * log n

O(n) means an operation take time less than c*n

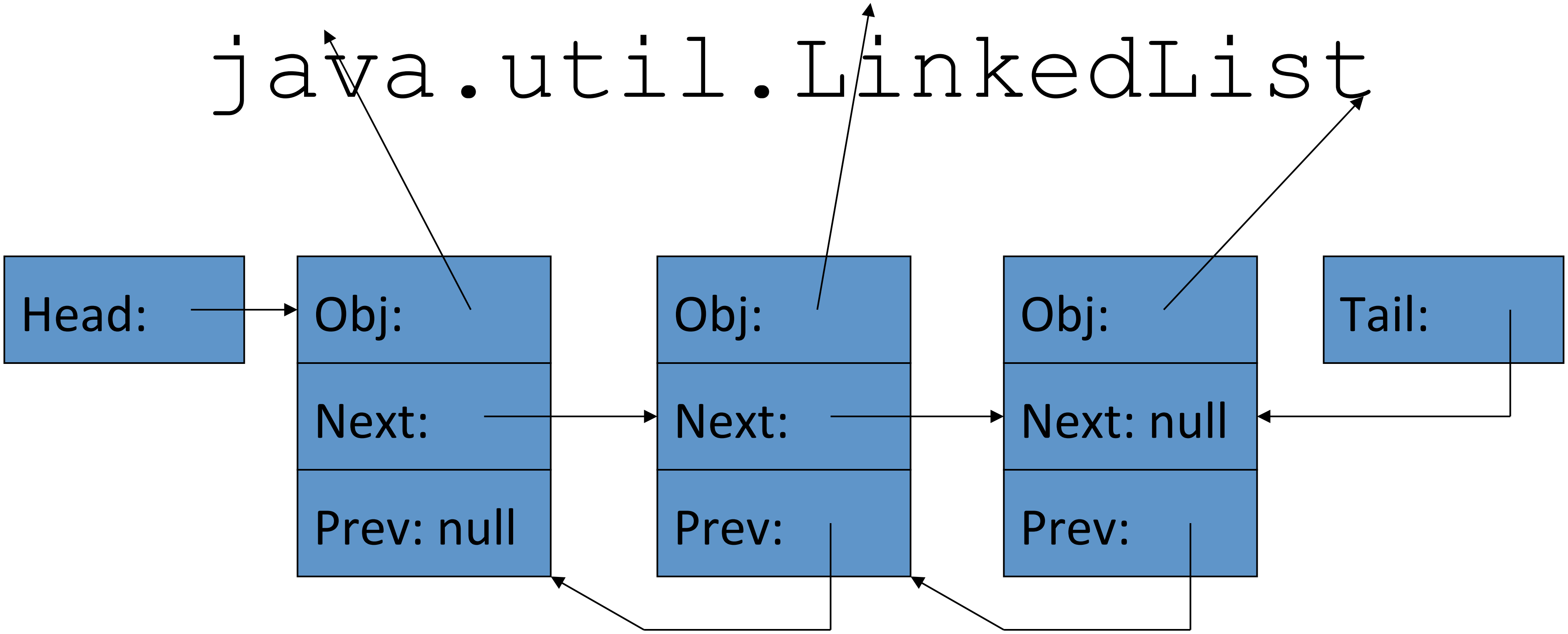
O(n log n) means an operation take time less than c * n * log n

O(n^2) means an operation take time less than c* n^2

Can O(*n*) be faster than O(1)?

- What about when *c* is large for O(1)?
- For example consider:
  - A O(1) insert @ front of list: t = 1,000 us
  - A O(*n*) insert @ front of list: t = 10 * *n* us
- For large *n*, O(1) will be faster than O(*n*)
  - In this example: for *n* > 100

`java.util.LinkedList`



- – Constant-time O(1) to append-to-back, insert-at-front, remove-from-front, getFirst, getLast
- – Linear-time O(*n*) to manipulate the middle

Performance Comparisons

	Append After Last	Insert Before First	Lookup by Position	Lookup by Value	Remove Last	Remove First
Array ArrayList	O(1)*	O(*n*)	O(1)	O(*n*)	O(1)	O(*n*)
LinkedList	O(1)	O(1)	O(*n*)	O(*n*)	O(1)	O(1)

* On average this operation will be constant O(1) time.

The List Interface

- Specifies common methods that must be implemented by all lists
- Implemented by ArrayList, Vector, LinkedList, and others
- All part of the Java Collections Framework
  - http://docs.oracle.com/javase/tutorial/collections

Java Interfaces

- We will use many interfaces in the upcoming lectures
- A Java Interface allows you to specify methods that must be implemented by a class
- You can then use references to the Interface, knowing the methods will be available, even though you don't know the specific class used
- This is Java's answer to multiple inheritance

Generics in Java

- Java allows you to specify a class that manipulates instances some type <T>
- Java really uses type erasure
  - There is only one class file for a generic class
  - The classes code doesn't really know what type it's manipulating
  - The compiler checks the types where the class is used
  - Runtime just processes **`Object`**s
- See

  `docs.oracle.com/javase/tutorial/java/generics`

Examples of Generics

- Example generic class:
  - ArrayList
- Example generic interfaces:
  - List, Comparable

Example uses of Lists

- StringList.java

(Compare with StringArray.java)

Before Generics

- In Java 4 (and before)
  - The Collections Framework stored Objects
  - You could put any object into a List
  - When you took something out of a List...
    - You needed to cast it back into your type
    - You could check types using **`instanceof`** operator
- Example:
  - StringList4.java

Outline

✓Questions

✓Lists

✓Interfaces & Generics

⇢Maps

Sorting, again

Autoboxing

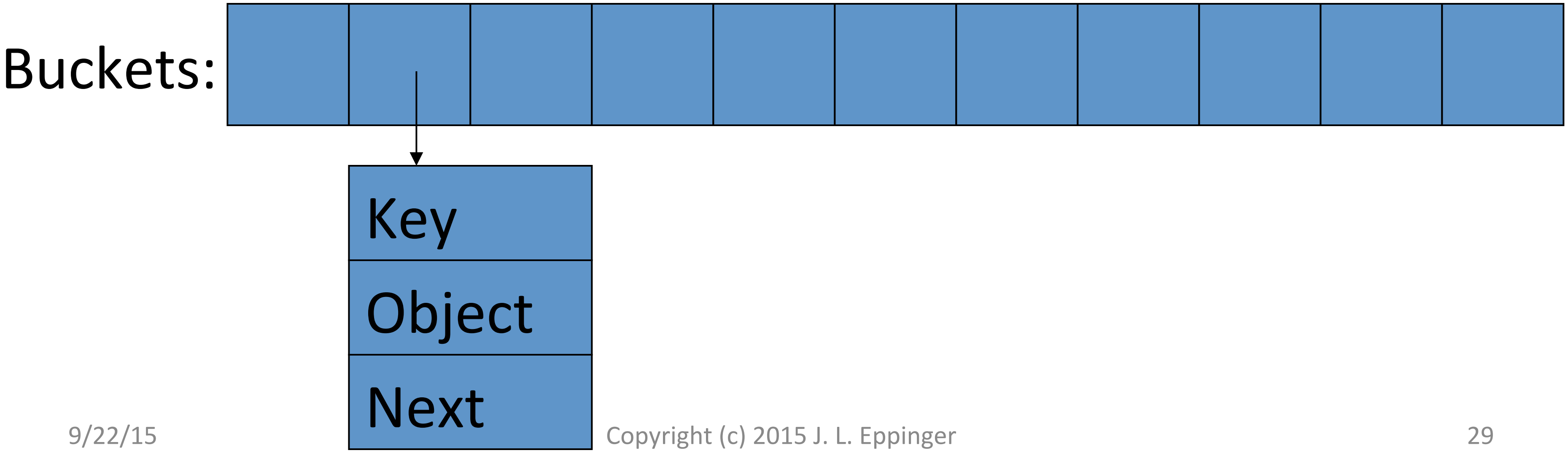
HW#5

Hash Functions

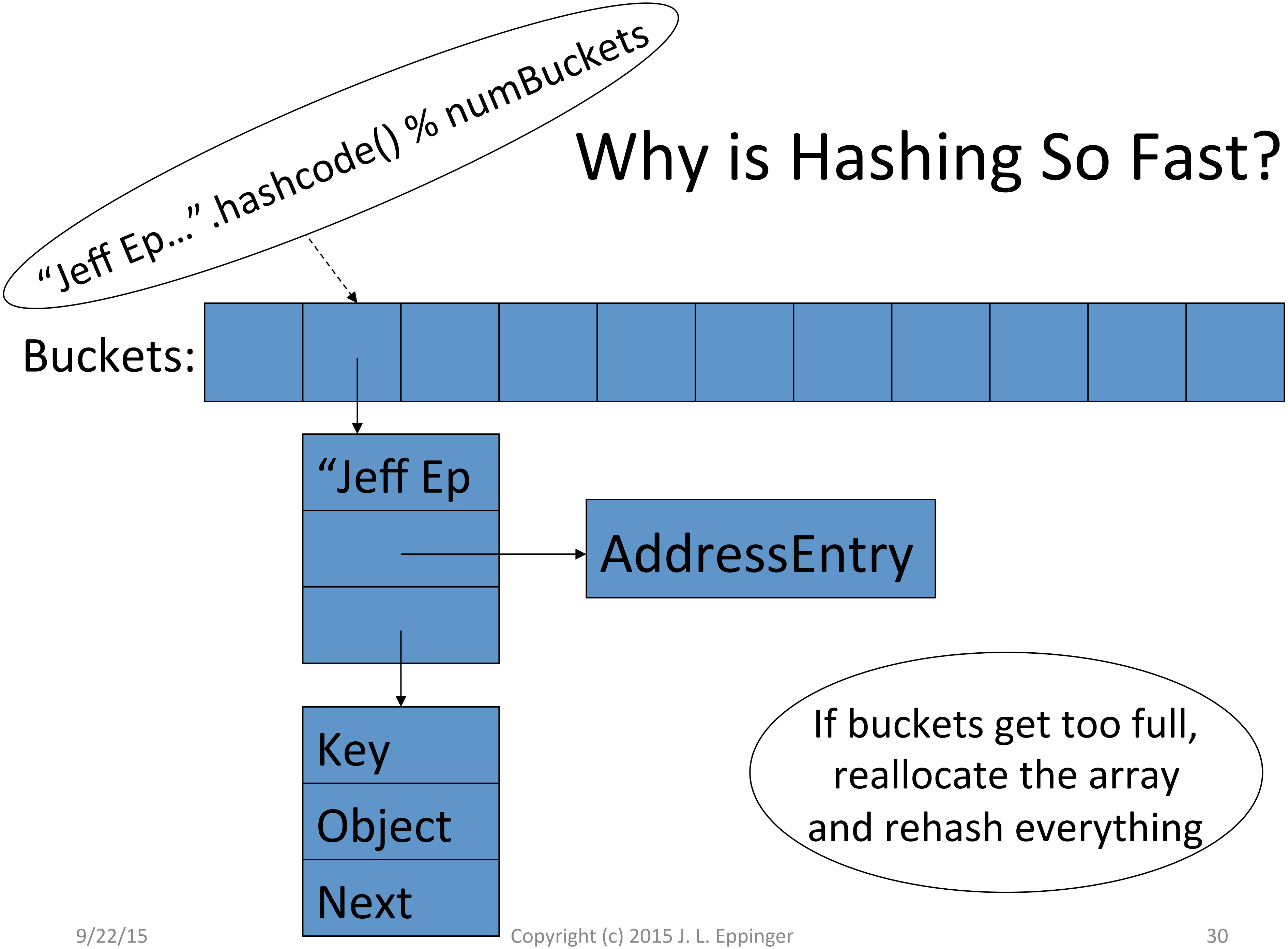
- Compress data into a number (usually an int)
- Two different inputs to should generally have two different outputs
- Example
  - StringHash.java
  - IntHash.java

java.util.HashMap

- Like an ArrayList, but elements are accessed by search key (not element number)
- Constant time O(1) for insert, delete, lookup anywhere
  - But no order to the elements



Why is Hashing So Fast?



If buckets get too full, reallocate the array and rehash everything

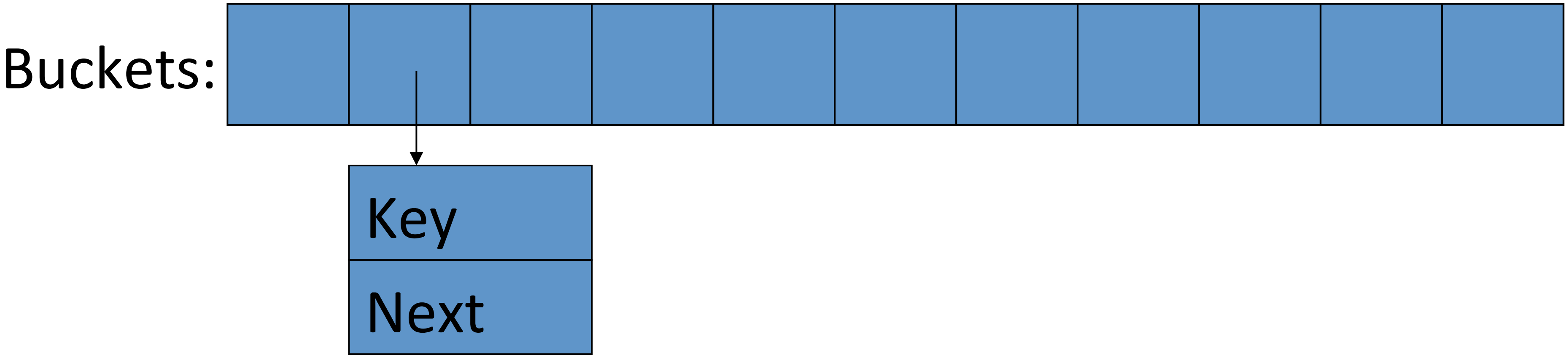
Bottom Line on HashMaps

- Optimized for access to any element, given the key (which is any object)

Example

- HashMapLookup.java

java.util.HashSet



- Just like a HashMap, but no mapping from key to object. It simply tells if the key is in the set.

Interfaces

- Note that there are Map and Set interfaces

Performance Comparisons

	Append After Last	Insert Before First	Lookup by Position	Lookup by Value	Remove Last	Remove First
ArrayList	O(1)*	O(n)	O(1)	O(n)	O(1)	O(n)
LinkedList	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
HashSet HashMap	Add: O(1)		N/A	O(1)	Remove: O(1)	

* On average this operation will be constant O(1) time.

Combining Data Structures!

	Append After Last	Lookup by Position	Lookup by Value	Remove First
HashSet HashMap	N/A	N/A	O(1)	N/A
LinkedHashSet LinkedHashMap	O(1)	O(*n*)	O(1)	O(1)

Outline

✓Questions

✓Lists

✓Interfaces & Generics

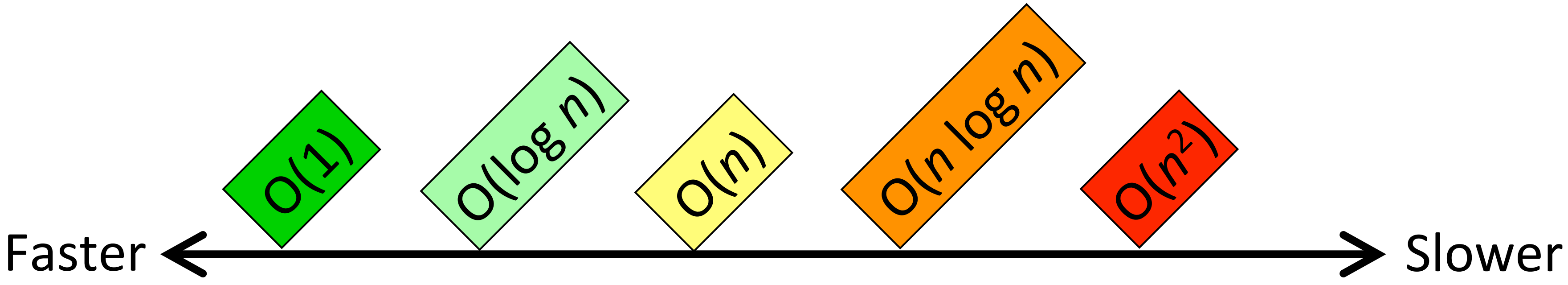
✓Maps

⇢Sorting, again

Autoboxing

HW#5

Fast and Slow



Faster

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

Slower

n is the number of elements

c is some constant

O(1) means an operation take time less than *c*

O(log *n*) means an operation take time less than $c * \log n$

O(*n*) means an operation take time less than $c*n$

O(*n* log *n*) means an operation take time less than $c * n * \log n$

O(n^2) means an operation take time less than $c* n^2$

Example: Sorting

- Selection Sort Performance – O(n^2)
- java.util.Arrays.sort – O(n log n)
- java.util.Collections.sort – O(n log n)
- Examples:
  - StringSort.java
  - SelectionSort.java
  - Sort.java

java.util.Comparator

- Pass a Comparator to sort to provide an alternative ordering
- It's a Java Interface
- Example:
  - StringSort2.java

Array Variants

- Arrays, ArrayLists, Vectors
  - All implement operations using a contiguous storage
  - Constant-time O(1) to append-to-back, get, set
  - Linear-time O(*n*) to insert-at-front, remove-from-front, search

Outline

✓Questions

✓Lists

✓Interfaces & Generics

✓Maps

✓Sorting, again

⇢Autoboxing

HW#5

ArrayList<int>?

- What if you want an ArrayList of ints?
- Generics only work for Objects
- Use the Integer wrapper class
  - Not only does this class provide helpers for ints
  - Instances store an int in an Object
  - Similar for other primitive types

```
int myInt = Integer.parseInt(args[0]);
List<Integer> list = new ArrayList<Integer>();
list.add(new Integer(myInt));
…
Integer x = list.get(0);
int i = x.intValue();
```

Autoboxing

- Writing the code to put your ints in Integers is a hassle
  - Same for other primitives
- In Java 5, Java will automatically convert between primitives and their Object wrapper classes
  - When passing parameters or returning values
  - In assignment or math expressions

ArrayList<Integer>

- Just declare `ArrayList<Integer>`
- Put in and take out `int`s
- Autoboxing automatically does the conversions

Outline

✓Questions

✓Lists

✓Interfaces & Generics

✓Maps

⇢Sorting, again

⇢Autoboxing

⇢HW#5

HW#5

- Will be posted today!
- You’ll be using data structures
- Thursday we will talk about the file loading part
- Friday we will discuss strategy
- But, it’s due on Monday, so get started

Sample Final Exam Questions

- Compare the use of ArrayLists, LinkedLists, and Arrays? What are the advantages of each?
- How fast is selection sort? How fast is the sort provided by java.util.Arrays?
- What is a comparator?
- What is autoboxing? Why is it useful in Java?
- What are Java Generics? What are the advantages of using generic classes? What did Java programmers do before we had generic classes?