Announcements
Today’s topics

- Quicksort
- More with Java Generics
public void mergeSort( int[] toSort, int start, int end )
{
    if ( start < end ) {
        int mid = start + ((end - start) / 2);
        mergeSort( toSort, workArray, start, mid );
        mergeSort( toSort, workArray, mid+1, end );
        merge( toSort, workArray, start, mid, end );
    }
}

This mergeSort works!!
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Merge Sort: An Example

About how instructions are executed at each split?
A. 1  
B. log(N)  
C. N  
D. N^2

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
How many merge steps are there?
A. \( \log(N) \)
B. \( N \)
C. \( N^2 \)

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
How many comparisons are made at each merge? (total)
A. \( \log(N) \)
B. \( N \)
C. \( N^2 \)
D. Other

Merge Sort follows a series of splitting steps with a series of merging steps to produce sorted partitions.
Running time for Merge sort:
\[ \log(N) \times 1 + \log(N) \times N = O(N \times \log(N)) \]
Quicksort: Another magical (recursive) algorithm

| 12 | 4 | 9 | 14 | 15 | 8 | 19 | 2 |

Select a **pivot** element:

| 12 | 4 | 9 | 14 | 15 | 8 | 19 | 2 |

“Partition” the elements in the array (**smaller**, **pivot**, **larger**)

| 8 | 4 | 9 | 2 | 12 | 15 | 19 | 14 |

Magically sort the smaller elements and the larger elements (Quicksort)

| 2 | 4 | 8 | 9 | 12 | 15 | 19 | 21 |
Quicksort: Another magical (recursive) algorithm

Select a **pivot** element:

```
12  4  9  14  15  8  19  2
```

"Partition" the elements in the array (**smaller**, **pivot**, **larger**)

```
8  4  9  2  12  15  19  14
```

We won’t cover how the partition step works, but see if you can figure it out! (or Google it)
Quick Sort: Using a “good” pivot

How many levels will there be if you choose a pivot that divides the list in half?
A. 1
B. $\log(N)$
C. $N$
D. $N\log(N)$
E. $N^2$
Quick Sort: Using a “good” pivot

If the time to partition on each level takes \( N \) comparisons, how long does Quicksort take with a good partition?

A. \( O(1) \)
B. \( O(\log(N)) \)
C. \( O(N) \)
D. \( O(N\log(N)) \)
E. \( O(N^2) \)
Quick Sort: Using a “good” pivot

If the time to partition on each level takes $N$ comparisons, how long does Quicksort take with a good partition?
A. $O(1)$
B. $O(\log(N))$
C. $O(N)$
D. $O(N \log(N))$
E. $O(N^2)$

Space complexity: $O(\log_2 n)$ for the runtime stack activation records
Which of these choices would be the worst choice for the pivot?

A. The minimum element in the list
B. The last element in the list
C. The first element in the list
D. A random element in the list
Quick sort with a bad pivot

Space complexity: $O(n)$ for the activation records

If the pivot always produces one empty partition and one with $n - 1$ elements, there will be $n$ levels, each of which requires $O(n)$ comparisons: $O(n^2)$ time complexity
Which of these choices is a better choice for the pivot?

A. The first element in the list
B. A random element in the list
C. They are about the same
Java Generics
Life before Generics

```java
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.get(0);
```

Element access needed casting.
Using Generics

List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.get(0);

No casting required. Cleaner code.
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));
.....
.....
String s = (String) myIntList.get(0);

A. YES
B. NO
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));

Does this compile?
YES! (may be with warnings)

String s = (String) myIntList.get(0);

What happens when you run it?
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));
......
......
String s = (String) myIntList.get(0);

Does this compile?
YES! (may be with warnings)

What happens when you run it?
ClassCastException!
Using Generics

List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
......
......
String s = (String) myIntList.get(0);

Does this compile?
Using Generics

List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
......
......
String s = (String) myIntList.get(0);

Does this compile?

NO!

Generics ⇒ **Stronger type checking at compile time** ⇒
No Runtime exceptions
Java Generics

List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());

What does this snippet print?

A. True
B. False

This returns type 'Class' not a string.
Java Generics

List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());

What does this snippet print?

A. True

B. False

Behavior is same for all classes. Hence the name ‘generic’.

This returns type ‘Class’ not a string. In this case, returns class ArrayList.

After compilation checks, the generic types are ‘erased’ and replaced by the first “bounding” superclass.
Java Generics - Erasure

```java
public class Node<T> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
}

public class Node {
    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
}
```
Non-covariance of Generics

```java
Collection<Animal> animalList;
Collection<Dog> dogList = new LinkedList<Dog>();
animalList = dogList;
...
...
animalList.add(0, new Cat());
Dog d = dogList.get(0);
```

- Java parameterized types are **not covariant**! A collection of Dogs is NOT a collection of Animals!

- This means that a Collection (or any class) parameterized by a subclass cannot be assigned to a Collection parameterized by the superclass.
Another example

```java
10    // Display all Shape objects in the given Collection.
11    // Call their display() instance method to do that.
12    static void displayShapes( Collection<Shape> collection ){
13        for ( Shape shape : collection )
14            shape.display();
15    }
16
17    Collection<Shape> shapes = new LinkedList<Shape>();
18    shapes.add( new Circle( 5.0 ) );
19    shapes.add( new Rectangle( 4.5, 21.2 ) );
20    displayShapes( shapes );
21
22    Collection<Circle> circles = new LinkedList<Circle>();
23    circles.add( new Circle( 5.0 ) );
24    circles.add( new Circle( 15.0 ) );
25    circles.add( new Circle( 25.0 ) );
26    displayShapes( circles );    // ERROR!
```
Yet another example

```
LinkedList<Integer> intList = new LinkedList<Integer>();
...
...
Collection<Integer> c = intList;  // Will this compile?
```

A. Yes

B. No
Yet another example

LinkedList<Integer> intList = new LinkedList<Integer>();
...
...
Collection<Integer> c = intList;

Will this compile?

A. Yes

B. No
Wildcards
Wildcards

Problem: The method should accept a Collection of any subclass of Shape:

```java
static void displayShapes(_______ listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
```

Java provides a flexible type – the wildcard – ‘?’
Unbounded wildcard – ‘?’

- `<?>` means any type.
- `Collection<?>` is a collection of any type.

```java
static void displayShapes(Collection<?> listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
```

Does this solve our problem?
Bounded wildcards

Problem: The method should accept a Collection of any subclass of Shape:

- Unbounded wildcards do not help as they accept a Collection of any type.
- What we want is “any type that extends Shape”
Bounded wildcards

Problem: The method should accept a Collection of any subclass of Shape:

- Unbounded wildcards do not help as they accept a Collection of any type.
- What we want is “any type that extends Shape”

```java
<? extends Shape>
```

Accept any type that is ‘upper bounded’ by Shape
Bounded wildcards

static void displayShapes(Collection<? extends Shape> listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
Problem: addAll should accept collections that contain any type that ‘is-a’ E.

```java
public abstract class AbstractCollection<E> implements Collection<E> {
    // Add all the elements of the argument Collection to this Collection
    public boolean addAll(_______ c) {
        // Add all the elements of the argument Collection to this Collection
        //Collection to this Collection
        public boolean addAll(_______ c) {

A. Collections<E>
B. Collections<?>
C. Collection<? extends E>
D. Collection<? super E>
E. More than one / None
```
public <T extends Comparable<? super T>> void sort(List<T> list);

T must be a type that implements the Comparable interface

What does it have to be able to compare itself to? Anything that is a T, and all SUPER classes of T

Why superclasses and not subclasses?
Generics... can you do:

Which of the following compile?

- `Collection<List> c = new LinkedList<List>();`
- `LinkedList<List> myL = new LinkedList<List>();`
- `LinkedList<List> myL = new LinkedList<ArrayList>();`
- `LinkedList<? extends List> myL = new LinkedList<ArrayList>();`
- `LinkedList<? super List> myL = new LinkedList<ArrayList>();`
- `LinkedList<? super List> myL = new LinkedList<Collection>();`
- `LinkedList<Collection> myL = new LinkedList<Collection>();`
- `myL.add( new ArrayList() );`
Generics... can you do:

- Does this compile? If not, why not? And how do you fix it?

```java
public static void myMethod( Collection<List> arg ) { ... }
```
```
// in main
Collection<LinkedList> myL = new Collection<LinkedList>();
myMethod( myL );
```