A Smorgasbord:

(Thanks for the picture, Wikipedia.)

Duke Comp Sci. is great!

(and some review)
A review: `.equals()`

Asks: “Do these two objects have the same value?”

```java
public class Pumpkin {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
}
```

Ron Wallace may have just set the world pumpkin record.
A review: `.equals()`

Asks: “Do these two objects have the same value?”

```java
class Pumpkin {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }

    public boolean equals(Object other) {
        return this.myMass == other.myMass && this.myGrowerName.equals(other.myGrowerName);
    }
}
```

`.equals` takes an `Object` as its argument for historical reasons. We’d rather it didn’t.

Ron Wallace may have just set the world pumpkin record. He would like to prove that his pumpkin has no equal.
A review: .equals()

Asks: “Do these two objects have the same value?”

```java
public class Pumpkin {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }

    public boolean equals(Object other) {
        // Implementation
    }
}
```

Ron Wallace may have just set the world pumpkin record. He would like to prove that his pumpkin has no equal.

http://goo.gl/v9zat

.equals takes an Object as its argument for historical reasons. We’d rather it didn’t.
A review: `.equals()`

Asks: “Do these two objects have the same value?”

```java
public class Pumpkin {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }

    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other.getClass() != getClass()) {
            return false;
        }
        Pumpkin p = (Pumpkin)other;
        return myMass == p.myMass &&
               myGrowerName.equals(p.myGrowerName);
    }
}
```

Common case: check all of the instance variables.

Ron Wallace may have just set the world pumpkin record. He would like to prove that his pumpkin has no equal.
A review: `.compareTo()`

Asks: “Which one of these objects is ‘bigger’ than the other?”

Ron Wallace may have just set the world pumpkin record. He has proven that his pumpkin has no equal. ✔

Now he wants to prove that his pumpkin is the best!

```java
public class Pumpkin implements Comparable<Pumpkin>{
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals hidden for space's sake.
    public int compareTo(Pumpkin other) {
        } } 
```

http://goo.gl/gKr0X

Note “implements Comparable<Pumpkin>”. This is why we don’t have to send `.compareTo` an Object; we wish `.equals` did this...
A review: `.compareTo()`

Asks: “Which one of these objects is ‘bigger’ than the other?”

```java
public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }

    // equals hidden for space's sake.
    public int compareTo(Pumpkin other) {
        if (myMass < other.myMass) {
            return -1;
        }
        if (myMass > other.myMass) {
            return 1;
        }
        return myGrowerName.compareTo(other.myGrowerName);
    }
}
```

Now we can use `Arrays.sort` on arrays of Pumpkins, `Collections.sort` on Lists of Pumpkins, and use Pumpkins in TreeSet and TreeMap.

Ron Wallace may have just set the world pumpkin record. He has proven that his pumpkin has no equal.

Now he wants to prove that his pumpkin is the best!

Note “implements Comparable<Pumpkin>”. This is why we don’t have to send `.compareTo` an Object; we wish `.equals` did this...
A review: .compareTo()

Asks: “Which one of these objects is ‘bigger’ than the other?”

```
public class Pumpkin implements Comparable<Pumpkin>{
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals hidden for space's sake.
    public int compareTo(Pumpkin other) {
        if (myMass < other.myMass) {
            return -1;
        }
        if (myMass > other.myMass) {
            return 1;
        }
        return myGrowerName.compareTo(other.myGrowerName);
    }
}
```

Now we can use Arrays.sort on arrays of Pumpkins, Collections.sort on Lists of Pumpkins, and use Pumpkins in TreeSet and TreeMap.

Note “implements Comparable<Pumpkin>”. This is why we don’t have to send .compareTo an Object; we wish .equals did this...

Ron Wallace may have just set the world pumpkin record.
He has proven that his pumpkin has no equal.
He has proven that his pumpkin is the best!
A review: .hashCode()

Less obvious. Turns a your object into an integer.

public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals & compareTo hidden for space's sake.
    public int hashCode() {
        // Nothing to do with breakfast foods
    }
}

Ron Wallace may have just set the world pumpkin record.
He has proven that his pumpkin has no equal.
He has proven that his pumpkin is the best!

Now he wants to hash his pumpkin.
A review: .hashCode()

Less obvious. Turns a your object into an integer.

public class Pumpkin implements Comparable<Pumpkin>{
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals & compareTo hidden for space's sake.
    public int hashCode() {
        // Nothing to do with breakfast foods
    }
}

But wait!

Ron Wallace may have just set the world pumpkin record.
He has proven that his pumpkin has no equal. ✔
He has proven that his pumpkin is the best! ✔
Now he wants to hash his pumpkin.
A review: .hashCode()

Less obvious. Turns your object into an integer.

Ron Wallace may have just set the world pumpkin record. He has proven that his pumpkin has no equal. ✔

He has proven that his pumpkin is the best! ✔

Now he wants to hash his pumpkin. (Nothing to do with breakfast foods)

public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals & compareTo hidden for space's sake.
    public int hashCode() {
    }
}

hashCode rules:
1. Depends only on the instance variables.
2. If a.equals(b), then a.hashCode() == b.hashCode().
3. If !a.equals(b), then a.hashCode might == b.hashCode.
4. All built-in Object types have a .hashCode. It’s a handy building block for your own hashCodes.
5. If !a.equals(b), a.hashCode() should try not to == b.hashCode.

Very important.

HashSet & HashMap will be faster if it isn’t. More details later!
A review: `.hashCode()`

Less obvious. Turns a your object into an integer.

```java
public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals & compareTo hidden for space's sake.

    public int hashCode() {
        // Nothing to do with breakfast foods
    }
}
```

Ron Wallace may have just set the world pumpkin record.
He has proven that his pumpkin has no equal.

He has proven that his pumpkin is the best!

Now he wants to hash his pumpkin.
(Nothing to do with breakfast foods)
A review: `.hashCode()`

Less obvious. Turns a your object into an integer.

```java
public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
    }
    // equals & compareTo hidden for space's sake.
    public int hashCode() {
        return myGrowerName.hashCode() +
            new Integer(myMass).hashCode();
    }
}
```

hashCode rules:
1. Depends only on the instance variables.
2. If a.equals(b), then a.hashCode() == b.hashCode().
3. If !a.equals(b), then a.hashCode() might == b.hashCode.
4. All built-in Object types have a .hashCode. It's a handy building block for your own hashCodes.
5. If !a.equals(b), a.hashCode() should try not to == b.hashCode().

Ron Wallace may have just set the world pumpkin record.
He has proven that his pumpkin has no equal.
He has proven that his pumpkin is the best!
(Nothing to do with breakfast foods)
In sum:

```java
public class Pumpkin implements Comparable<Pumpkin> {
    private int myMass;
    private String myGrowerName;
    private int myHashCode;

    public Pumpkin(int mass, String grower) {
        myMass = mass;
        myGrowerName = grower;
        computeHashCode();
    }

    public boolean equals(Object other) {
        if (other == null) {
            return false;
        }
        if (other.getClass() != getClass()) {
            return false;
        }
        Pumpkin p = (Pumpkin) other;
        return myMass == p.myMass &&
               myGrowerName.equals(p.myGrowerName);
    }

    public int compareTo(Pumpkin other) {
        if (myMass < other.myMass) {
            return -1;
        }
        if (myMass > other.myMass) {
            return 1;
        }
        return myGrowerName.compareTo(other.myGrowerName);
    }

    private void computeHashCode() {
        myHashCode = myGrowerName.hashCode() +
                     new Integer(myMass).hashCode();
    }

    public int hashCode() {
        return myHashCode;
    }
}
```
```
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedListNode node = new LinkedListNode("great!", null);
        node = new LinkedListNode("is", node);
        node = new LinkedListNode("Sci.", node);
        node = new LinkedListNode("Comp.", node);
        node = new LinkedListNode("Duke", node);
    }
}
```

```
public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}
```
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedListNode node = new LinkedListNode("great!", null);
        node = new LinkedListNode("is", node);
        node = new LinkedListNode("Sci.", node);
        node = new LinkedListNode("Comp.", node);
        node = new LinkedListNode("Duke", node);
    }
}

Recursive data!

...suggests recursive algorithms.
void printList(LinkedListNode node) {

    // Base Case
    if (node == null) {
        return;
    }

    // Deal with this node
    System.out.println(node.getString());

    // Deal with the rest of the list
    printList(node.getNext());
}
void printList(LinkedListNode node) {
    if (node == null) {
        return;
    }
    Deal with this node
    Deal with the rest of the list
}

public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}
public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}

void printList(LinkedListNode node) {
    if (node == null) {
        return;
    }
    System.out.println(node.getString());
    printList(node.getNext());
}
void printList(LinkedListNode node) {
    if (node == null) {
        return;
    }

    System.out.println(node.getString());
    printList(node.getNext());
}

public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}
public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}

void printList(LinkedListNode node) {
    if (node == null) {
        return;
    }
    System.out.println(node.getString());
    printList(node.getNext());
}

http://goo.gl/6q1eI
void printList(LinkedListNode node) {
    if (node == null) {
        return;
    }

    System.out.println(node.getString());
    printList(node.getNext());
}

public class LinkedListNode {
    private String myString;
    private LinkedListNode myNext;
    public LinkedListNode(String s, LinkedListNode n) {
        myString = s;
        myNext = n;
    }
    public String getString() {
        return myString;
    }
    public LinkedListNode getNext() {
        return myNext;
    }
}

Snarf LinkDemo
http://goo.gl/6q1el

http://goo.gl/GNDhP
What values can we put in this square?
Snarf Sudoku

Implement recursiveHelper (in Sudoku)

http://goo.gl/A70fL