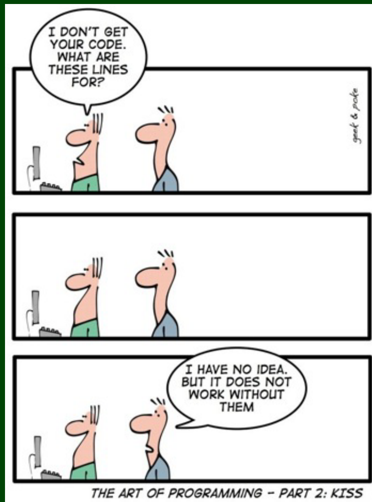
CSE 143X

Accelerated Computer Programming I/II

Comparable



Storing Multiple Choice Quizzes

The text files:

- Each text file corresponds to answers for a multiple choice quiz.
- Each line contains one answer.
- For each quiz, answers.txt represents the correct answers.

MCQuiz Class

```
1  public class MCQuiz {
2     private String studentName;
3     private String quizName;
4     private List<String> correctAnswers;
5     private List<String> studentAnswers;
6
7     public MCQuiz(String filename) throws FileNotFoundException { ... }
8
9     public String getStudent() { ... }
10    public String getName() { ... }
11    public int numberCorrect() { ... }
12 }
```

We would like to do the two following tasks:

1. Print out the quizzes in worst-to-best order (e.g. sort the quizzes)
2. Collect all quizzes of each particular student together and display them (still from worst-to-best)

Sorting Integers and Strings

Last lecture, we sorted the characters of a string. Let's sort more:

Sorting An Integer List

```
1  public static void sortIntList(List<Integer> list) {
2     for (int i = 0; i < list.size(); i++) {
3        int minIndex = i;
4        for (int j = i; j < list.size(); j++) {
5           if (list.get(j) < list.get(minIndex)) {
6              minIndex = j;
7           }
8        }
9        int temp = list.get(minIndex);
10       list.set(minIndex, list.get(i));
11       list.set(i, temp);
12    }
13 }
```

Sorting A String List

```
1  public static void sortStringList(List<String> list) {
2     for (int i = 0; i < list.size(); i++) {
3        int minIndex = i;
4        for (int j = i; j < list.size(); j++) {
5           if (list.get(j) < list.get(minIndex)) {
6              minIndex = j;
7           }
8        }
9        String temp = list.get(minIndex);
10       list.set(minIndex, list.get(i));
11       list.set(i, temp);
12    }
13 }
```

Sorting A String List

```
1 if (list.get(j) < list.get(minIndex)) {
2    minIndex = j;
3 }
```

compareTo

Strings have a method called `compareTo` that works like < does on ints.

If we have two strings:

`String hello = "hello"` and `String bye = "bye"`

To do the test "`hello < bye`", we do the following:

1. Write what we want: `hello < bye`
2. Subtract the right from both sides: `hello - bye < 0`
3. Replace the subtraction with `compareTo`:

   `hello.compareTo(bye) < 0`

That's it!

Sorting A String List

```
1 if (list.get(j).compareTo(list.get(minIndex)) < 0) {
2    minIndex = j;
3 }
```

Sorting A MCQuiz List

```
public static void sort(List<MCQuiz> list) {
   for (int i = 0; i < list.size(); i++) {
      int minIndex = i;
      for (int j = i; j < list.size(); j++) {
         if (list.get(j).numberCorrect() < list.get(minIndex).numberCorrect()) {
            minIndex = j;
         }
      }
      MCQuiz temp = list.get(minIndex);
      list.set(minIndex, list.get(i));
      list.set(i, temp);
   }
}
```

Strings were easier, because **they knew how to compare themselves**.

Implementing A compareTo

```
public int compareTo(MCQuiz other) {
// From above: list.get(j).numberCorrect() < list.get(minIndex).numberCorrect()
// Replacing:  this.numberCorrect() < other.numberCorrect()
// Converting: this.numberCorrect() - other.numberCorrect() < 0
   return this.numberCorrect() - other.numberCorrect();
}
```

Sorting An MCQuiz List

```
if (list.get(j).compareTo(list.get(minIndex)) < 0) {
   minIndex = j;
}
```

How do sort and TreeSet **KNOW** the ordering?

If you were implementing sort for a type T, what would you need to be able to do with T a and T b?

We would need to be able to COMPARE a and b

That's just an interface! Java calls it "Comparable".

Comparable

The Comparable interface allows us to tell Java how to **sort** a type of object:

```
1 public interface Comparable<E> {
2    public int compareTo(E other);
3 }
```

This says, "to be Comparable, classes must define compareTo".

Client Code to Print The Quizzes

```
List<MCQuiz> quizzes = createQuizzes(2);
// First, let's get a sorted list of the quizzes
Collections.sort(quizzes);
for (MCQuiz quiz : quizzes) {
   System.out.println(quiz);
}
```

This doesn't work, because Java doesn't know how to **sort** `MCQuizzes`.

`Comparable`

The `Comparable` interface allows us to tell Java how to **sort** a type of object:

```
public interface Comparable<E> {
   public int compareTo(E other);
}
```

This says, "to be `Comparable`, classes must define `compareTo`".

MCQuiz: Defining compareTo

Attempt #1

```
1 public class MCQuiz implements Comparable<MCQuiz> {
2    ...
3    public int compareTo(MCQuiz other) {
4       return this.numberCorrect() − other.numberCorrect();
5    }
```

This doesn't; work, because if we have a quiz where someone got $1/10$ and another where someone else got $1/5$, we treat them as the same.

Attempt #2

```
1 public class MCQuiz implements Comparable<MCQuiz> {
2    ...
3    public int compareTo(MCQuiz other) {
4       return (double)this.numberCorrect()/this.correctAnswers.size() −
5             (double)other.numberCorrect()/other.correctAnswers.size();
6    }
```

This won't even compile! We need to return an **int**.

int Fields

If we have a field `int x` in our class, and we want to compare with it, our code should look like:

```
1 public class Sample implements Comparable<Sample> {
2    public int compareTo(Sample other) {
3       return ((Integer)this.x).compareTo(other.x);
4    }
5 }
```

Object Fields

If we have a field `Thing x` in our class, and we want to compare with it, our code should look like:

```
1 public class Sample implements Comparable<Sample> {
2    public int compareTo(Sample other) {
3       return this.x.compareTo(other.x);
4    }
5 }
```

In other words, just use the existing `compareTo` on the field in the class!

Attempt #3

```
public class MCQuiz implements Comparable<MCQuiz> {
  ...
  public int compareTo(MCQuiz other) {
    Double thisPer = (double)this.numberCorrect()/this.correctAnswers.size();
    Double otherPer = (double)other.numberCorrect()/other.correctAnswers.size();
    return thisPer.compareTo(otherPer);
  }
```

This **still** doesn't work, because it doesn't take the **names** of the students into account.

In particular, if two students both get $1/10$ on a quiz, our `compareTo` method says "it doesn't matter which one goes first".

Attempt #4

```
public class MCQuiz implements Comparable<MCQuiz> {
  ...
  public int compareTo(MCQuiz other) {
    Double thisPer = (double)this.numberCorrect()/this.correctAnswers.size();
    Double otherPer = (double)other.numberCorrect()/other.correctAnswers.size();
    int result = thisPer.compareTo(otherPer);
    if (result == 0) { result = this.studentName.compareTo(other.studentName); }
    return result;
  }
```

This **still** doesn't work, but it's not as clear why. Let's try the second task.

What data structure should we use to group the quizzes? A Map!

Map Question: "Which quizzes were taken by this student?"
Keys: **Strings** (the student names)
Values: **Set<MCQuiz>** (all the quizzes that student took).

```
1  List<MCQuiz> quizzes = createQuizzes(2);
2  Map<String, Set<MCQuiz>> quizzesByStudent = new TreeMap<>();
3
4  // We want to loop over all the quizzes, adding them one by one
5  for (MCQuiz quiz : quizzes) {
6     String name = quiz.getStudent();
7     if (!quizzesByStudent.containsKey(name)) {
8        quizzesByStudent.put(name, new TreeSet<MCQuiz>());
9     }
10    quizzesByStudent.get(name).add(quiz);
11 }
12
13 // Now, we want to print out the quizzes student by student:
14 for (String student : quizzesByStudent.keySet()) {
15    System.out.println(student + ": " + quizzesByStudent.get(student));
16 }
```

The output looks like this:

```
OUTPUT
>> AdamBlank: [AdamBlank (quiz1): 1/11, AdamBlank (quiz0): 4/11]
>> BarbaraHarris: [BarbaraHarris (quiz1): 3/11, BarbaraHarris (quiz0): 4/11]
>> ChrisHill: [ChrisHill (quiz0): 3/11, ChrisHill (quiz1): 4/11]
>> JessicaHerna: [JessicaHernan (quiz1): 1/11, JessicaHernan (quiz0): 2/11]
>> TeresaHall: [TeresaHall (quiz0): 4/11]
```

Why does Teresa only have one quiz? **She scored the same on both of her quizzes and `compareTo` said they were the same!**

Final Attempt

```java
public class MCQuiz implements Comparable<MCQuiz> {
  ...
  public int compareTo(MCQuiz other) {
    Double thisPer = (double)this.numberCorrect()/this.correctAnswers.size();
    Double otherPer = (double)other.numberCorrect()/other.correctAnswers.size();
    int result = thisPer.compareTo(otherPer);
    if (result == 0) {
      result = this.studentName.compareTo(other.studentName);
    }
    if (result == 0) {
      result = this.quizName.compareTo(other.quizName);
    }
    return result;
  }
```

Lesson: When you write `compareTo`, make sure that

`a.compareTo(b) == 0` exactly when `a.equals(b)`

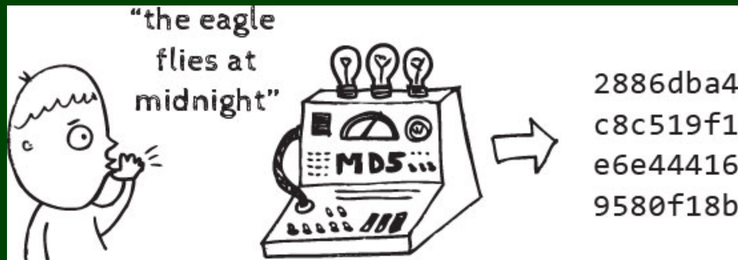
Some Comparable Tips

- Understand multi-level structures
- Use the most general interface as possible
- When implementing `compareTo`, make sure to use all the fields that make it different (to put another way: `a.compareTo(b) == 0` exactly when `a.equals(b)`)
- Remember that inside classes, you can look at the fields of other instances of that class

CSE 143X

Accelerated Computer Programming I/II

Hashing



Today, we will consider multiple new implementations of `IntSet`:

```java
public interface IntSet {
    public void add(int value);
    public void remove(int value);
    public boolean contains(int value);
}
```

A New Data Structure

Design a class `RangeSet` that represents a set which only allows numbers inside a **fixed range**.

You should have a constructor:

`RangeSet(max)`	This constructor initializes a new `RangeSet` which only allows elements between 0 (inclusive) and **max** (exclusive).

And the following **public** methods:

`add(`**val**`)`	Adds **val** to the `RangeSet` if it is a valid value and throws an `IllegalArgumentException` otherwise.
`remove(`**val**`)`	Removes **val** to the `RangeSet` if it is a valid value in the set and does nothing otherwise.
`contains(`**val**`)`	Returns true if **val** is in the `RangeSet` and false otherwise.

`add`, `remove`, and `contains` must all be $\mathcal{O}(1)$

```
1  public class RangeSet implements IntSet {
2     private boolean[] data;
3
4     public RangeSet(int max) { this.data = new boolean[max]; }
5
6     public void add(int value) {
7        if (value >= this.data.length || value < 0) {
8           throw new IllegalArgumentException();
9        }
10       this.data[value] = true;
11    }
12
13    public boolean contains(int value) {
14       if (value >= this.data.length || value < 0) {
15          return false;
16       }
17       return this.data[value];
18    }
19
20    public void remove(int value) {
21       if (value < this.data.length && value >= 0) {
22          this.data[value] = false;
23       }
24    }
25 }
```

In RangeSet, when we got the number n, we mapped it to the index n. What if we had a function that took an input and mapped it to an index?

Definition (HashCode)

A **hash code** is a function that takes in a piece of data and maps it to an array index.

If we have an array of size 8, consider the following hashcode:

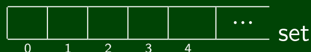
```
1 public int hashCode(int value) {
2    return value % 8;
3 }
```

Now, let's insert the following data: 1, 4, 13

HashSet Attempt #1

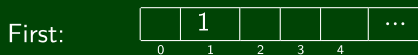

```
1  public class IntHashSet implements IntSet {
2     public final int DEFAULT_SIZE = 20;
3     public Integer[] data;
4
5     public IntHashSet() {
6        this.data = new Integer[DEFAULT_SIZE];
7     }
8
9     private int hashCode(int value) {
10       return value % data.length;
11
12    public void add(int value) {
13       this.data[hashCode(value)] = value;
14    }
15
16    public boolean contains(int value) {
17       return this.data[hashCode(value)] == value;
18    }
19
20    public void remove(int value) {
21       this.data[hashCode(value)] = null;
22    }
```

HashSet Attempt #1 Problem!



Consider the following insertions: 1, 21

First:

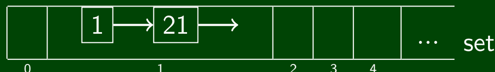


Then:

Uh oh! We've overwritten the one!

How can we fix this?

Instead of storing an integer, let's store a list of integers



String HashCode!


```
1  public int hashCode() {
2     int h = hash;
3     if (h == 0 && value.length > 0) {
4        char val[] = value;
5
6        for (int i = 0; i < value.length; i++) {
7           h = 31 * h + val[i];
8        }
9        hash = h;
10    }
11    return h;
12 }
```