Data Structures and Algorithms

XMUT-COMP 103 - 2025 T1

Sorting and Comparable Objects

Felix Yan

School of Engineering and Computer Science

Victoria University of Wellington

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting

- Sorting a list is easy:
  - Collections.sort(myList);
  - Pretty fast!

- But only if the items in the list are Comparable
  - Lists of String, Lists of numbers, …

- Sorting a Set is easy:
  - Use a TreeSet instead of a HashSet, or
  - Copy HashSet to a list and sort:

```
List<String> sortedVocab = new ArrayList<String>(vocab);     or   addAll(vocab);
Collections.sort(sortedVocab);
```

- But only if the items in the list are Comparable (eg, String, Integer, Double)

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting items

- What about Lists of items that aren't Comparable?

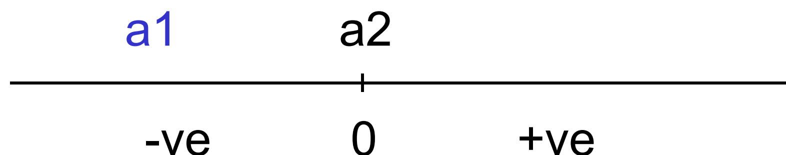
- Sort needs a method to compare two values and say which comes first
  - Comparable objects have such a method (called compareTo)
  - You can give sort such a method for non-Comparable objects
    - parameters are the two values
    - returns an int

      -1 if the first value comes earlier in the ordering (or any negative integer)

      1 if the second value comes earlier in the ordering (or any positive integer)

      0 if the two values are equal in the ordering

  - Collections.sort(myList, (Type a1, Type a2) -> { **if** (*a1 is before a2*) { **return** -1; }
    else if (*a1 is after a2*) { **return** 1; }
    else { **return** 0; } });



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting Students by City

- If classList is a List of Student objects, which have a getCity() method:

- Collections.sort(classList,

```
                (Student s1, Student s2) -> {
                    if      (s1.getCity().compareTo(s2.getCity()) < 0 )  { return -1; }
                    else if (s1.getCity().compareTo(s2.getCity()) > 0 )  { return  1; }
                    else                                                 { return  0; }
                } );
```

OR

- Collections.sort(classList,

```
                (Student s1, Student s2) -> { return ( s1.getCity().compareTo(s2.getCity() );  } );
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting Students by Country and then City

If classList is a List of Student objects, which have a getCountry() and getCity() methods:

```
Collections.sort(classList,
                 (Student s1, Student s2) -> {
                     if      (s1.getCountry().compareTo(s2.getCountry()) < 0 )  { return -1; }
                     else if (s1.getCountry().compareTo(s2.getCountry()) > 0 )  { return  1; }
                     else {   return ( s1.getCity().compareTo(s2.getCity()) ); }  // same country
                 } );
```

OR

```
Collections.sort(classList, this::compareByCountryCity);

public int compareByCountryCity(Student s1, Student s2) {
    if      (s1.getCountry().compareTo(s2.getCountry()) < 0 )  { return -1; }
    else if (s1.getCountry().compareTo(s2.getCountry()) > 0 )  { return  1; }
    else {   return ( s1.getCity().compareTo(s2.getCity()) ); }  // same country
}
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting items

- What about TreeSets or TreeMaps of items that aren’t Comparable?

- Need to give the Set (or Map) a method to compare two values and say which comes first

  - ```
    Set<Face> crowd = new TreeSet<Face>((Face f1, Face f2) -> {
                                   if (f1.size() < f2.size()){ return -1; } else….} ));
    ```

  - ```
    Set<Face> crowd = new TreeSet<Face>((Face f1, Face f2) -> {
                                   return (f1.size() - f2.size()));
    ```

  - ```
    Map<Person,Integer> counter = new TreeMap<Person,Integer>(
                                (Person p1, Person p2) -> {
                                  return ( p1.getName().compareTo(p2.getName()) );
                                });
    ```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei
    ```

Recap: the vocab map.

```
Map<String, Integer> vocab = new HashMap<String, Integer>();
Scanner sc = new Scanner(Path.of(filename));
while (sc.hasNext()){
    String word = sc.next();
    if (! vocab.containsKey(word)){
        vocab.put(word, 1);
    }
    else {
        vocab.put(word, (vocab.get(word)+1) );
    }
}

//  sort vocabulary by count and print out first 100 words     ??????
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting Vocab with a compare method

- Vocab
  - Map of [word-count] pairs
  - Sort by the count (largest first)
  - How do we get a list of the pairs?
  - Maps consist of key-value pairs of type Map.Entry<K, V>
    - Therefore, make a List<Map.Entry<K,V> >
  - How do we sort the List?
    - Using a comparison method:

```
List < Map.Entry<String, Integer> > list = new ArrayList< Map.Entry<String, Integer>
>(vocal.entrySet());

Collections.sort(list, (Map.Entry<String, Integer> p1, Map.Entry<String, Integer> p2) ->{
                    return ( p2.getValue()-p1.getValue() );
                }));
```

Is this the right way round?

[Run the VocabularyMap program]

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Sorting Comparable objects vs using a lambda

- Sorting Lists of Comparable values is MUCH easier.
- Strings, Double, Int are all Comparable.
- How could we make Student, Person, City, … Comparable?

Note: it would only work if there is just one obvious way or ordering the items.

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Making Objects that are Comparable

What does Comparable mean? How can you make your objects Comparable?

- The class must declare that it is Comparable

```
public class Student implements Comparable<Student>{…
```

- The class must have a compareTo(…) method:

```
public int compareTo(Student other){
    if ( this.sID < other.sID)        { return -1; }
    else if (this.sID > other.sID)) { return  1; }
    else                                    { return  0; }    });
```

Now can use sort directly on lists of students:
Collections.sort(classList);

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Making Classes usable with Collections

- Making a class Comparable makes it easier to use with sort, TreeSet, TreeMap,…
- There are other methods that may also be needed to make your class easy to use:
  - compareTo(…) [to make it Comparable]
  - toString() [to make it easy to print out objects]
  - equals(…) [to make everything work,
    needed if two different objects could be considered to be the same,
    like Strings]
  - hashCode() [to make HashSet and HashMap work]
- Part of making user-defined Classes usable with Collections
  - Note: compareTo, equals, and hashCode should be consistent!

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

public String toString(){…

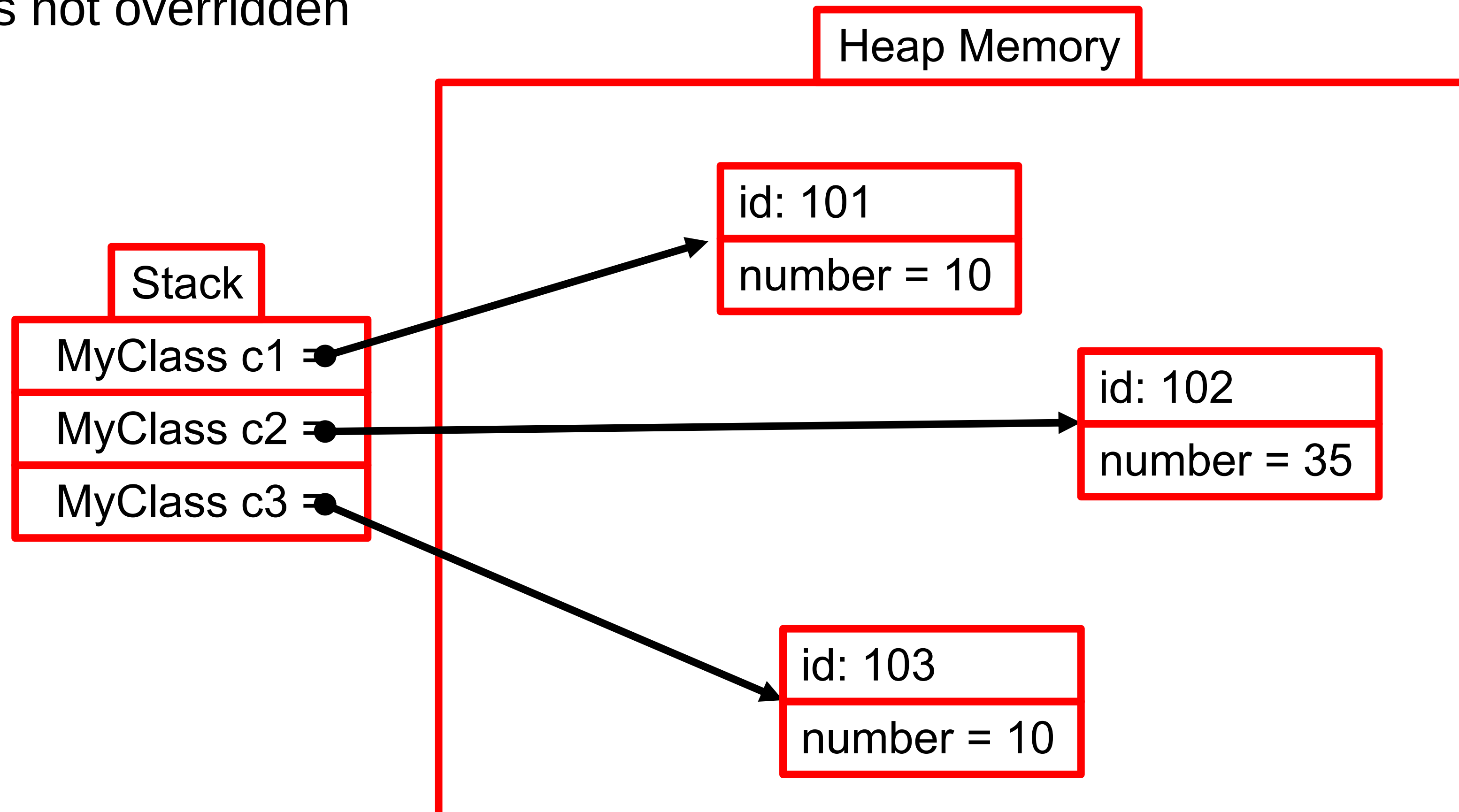
- If an object has a toString method, then print, printf, and println will automatically call the toString method to get a printable description of the object
- Every object has a toString() method – defined in the Object class.
  - The default toString() just returns the “ID” of the object (not very useful!)
- If a class defines its own (useful) toString() method, then objects of that class can be printed nicely.
- Always define a toString method that gives a useful description, at least for debugging!
  - The String returned should include all the important fields of the object.
- Lists, Sets, *etc* have nice toString() methods

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Potential problem with equals(…)

```
public class MyClass {
    private int number;

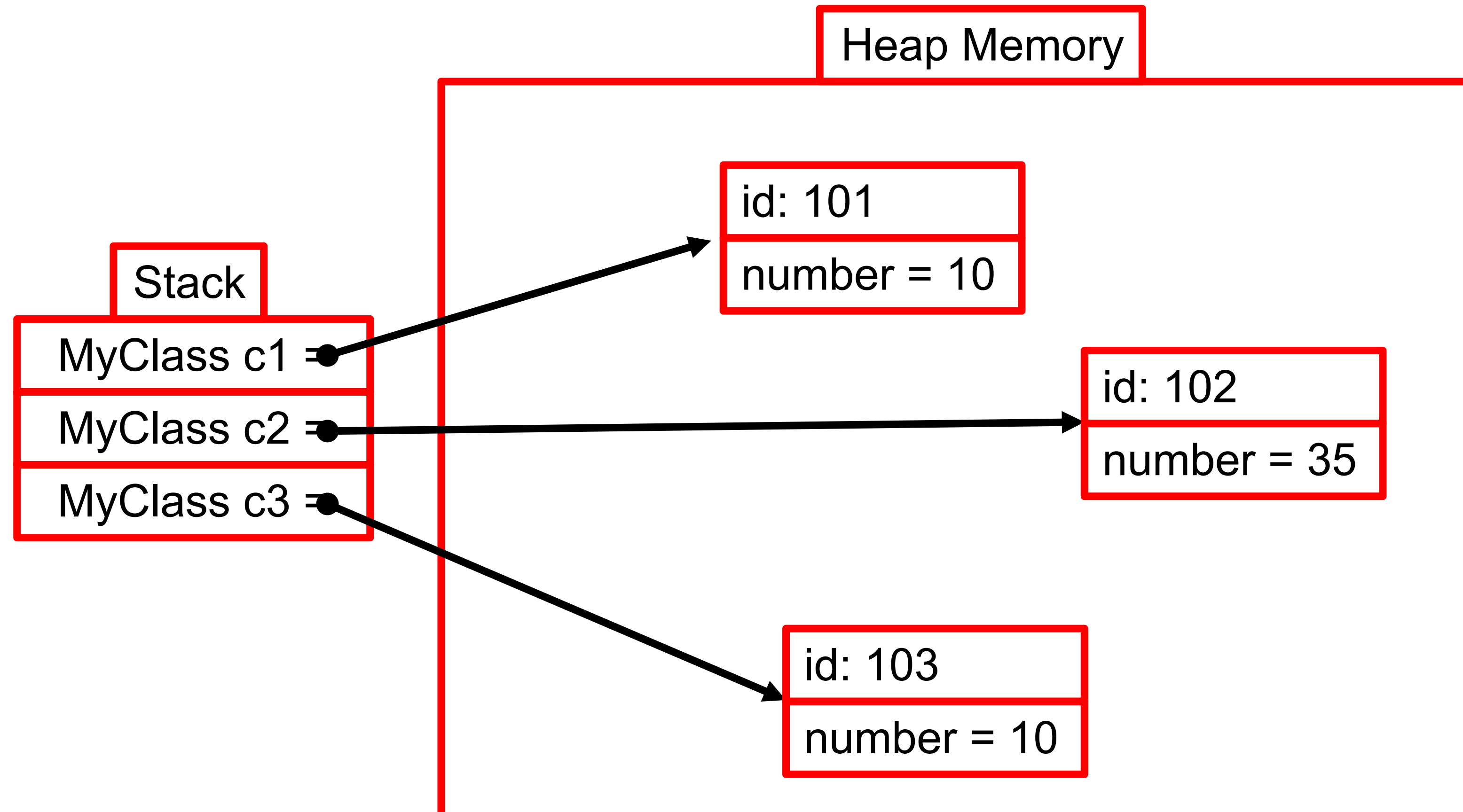
    //equals is not overridden
}
```



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Potential problem with equals(…)

Default equals() checks id of objects. Not the value(s) stored within the object.



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

public boolean equals(Object obj){…

- The Collection classes use the equals(..) method to determine if a value is the same as a value in the collection
- Every object has an equals(…) method – defined in the Object class.
  - The default equals(…) just does ==
    - just compares the references/IDs/pointers to see if the values are exactly the same Object.
    - doesn't look inside the objects to see if the fields are the same
- The String class (and the Collection classes) have useful equals(…)
- Always define an equals method in your classes if two different objects containing the same field values should be considered the same.
  - BUT
    - Need a compatible hashCode() method if using a HashMap
      (equal objects have same hashCode value)
    - Need a compatible compareTo if using a TreeMap
      (equal objects should have compareTo(..) → 0, and vice versa)

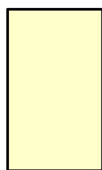
© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

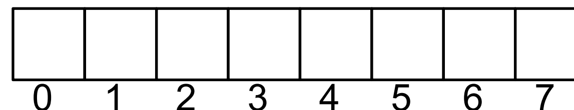
Designing an equals(…) method

- The equals method should reflect object identity:
  - If obj1.equals(obj2) then obj1 and obj2 should represent the same entity

- The equals method should not depend on fields that might change
  - Otherwise, objects are changing their identity,
  - two objects that are equal at one time may not be equal later.
  - (Same applies to the compareTo method!)

- The equals method needs to be consistent with the hashCode() method
  - two objects that are equal must have the same hashCode because hashSet and hashMap use both equals() and hashCode(0) to find values in the collection



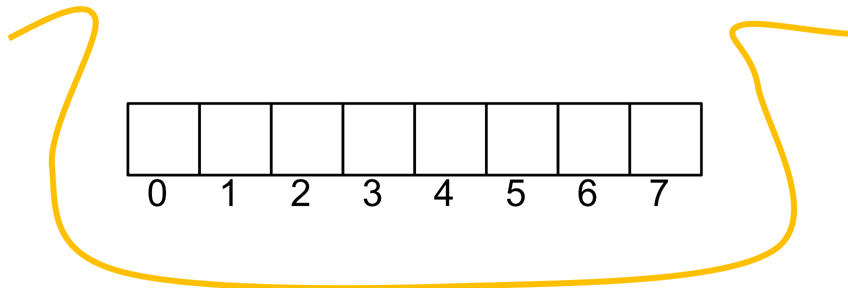


hashCode() % data.length

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Designing an equals(…) method

- The equals method should reflect object identity:
  - If obj1.equals(obj2) then obj1 and obj2 should represent the same entity
- The equals method should not depend on fields that might change
  - Otherwise, objects are changing their identity,
  - two objects that are equal at one time may not be equal later.
  - (Same applies to the compareTo method!)
- The equals method needs to be consistent with the hashCode() method
  - two objects that are equal must have the same hashCode because hashSet and hashMap use both equals() and hashCode() to find values in the collection



hashCode() % data.length to find the bucket

equals(..) to find the item in the bucket.

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Fields for equals/compareTo/hashCode:

- What fields should/should not be used for equals, compareTo and hashCode in:
  - a Student class (used in a student records system)
  - a Car class (used in a vehicle registration system)
  - a Shape class (used in a diagram editor)
  - an Earthquake class (used in the GNS earthquake record system)

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Designing equals/compareTo/hashCode

The methods should depend

- on fields that are sufficient to identify the entity represented by an object and distinguish it from other entities
- only on fields that will not be changed over time
    - (nice if the fields are declared final – so they *can't* be changed.)

If there are no such fields (eg, they might all change over time), or
Every time you create a new object of this type it should be considered unique

Then:

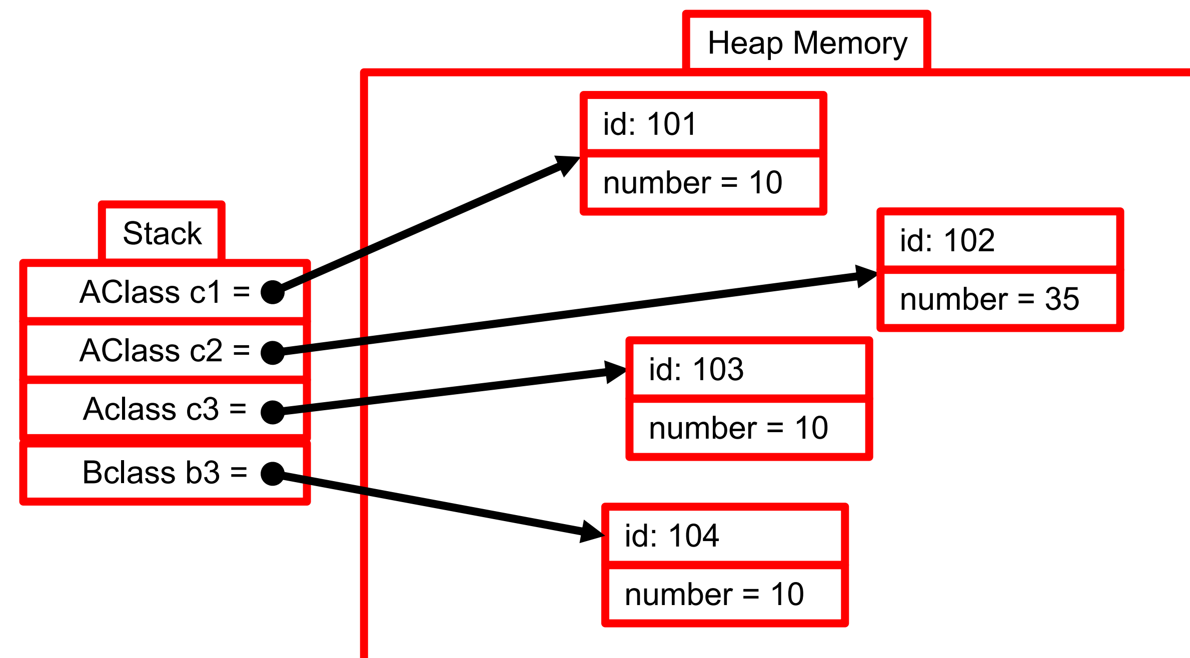
- use the default equals and hashCode which use the object reference/ID/pointer
- compareTo probably doesn't make sense.

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

public boolean equals(Object obj){…

- Standard way to override the equals method:
  - Check if obj is the same object as this. ! true
  - Check if obj is null ! false
  - Check if obj is the same type ! false if not
  - Cast obj to this type
  - Check if the fields are the same

```
public class AClass{
    .
    .
    public boolean equals(Object obj){
        if (this==obj)          { return true; }
        if (obj == null)        { return false; }
        if (! obj instanceof AClass ) { return false; }
        AClass other = (AClass) obj;
        // check if field values are the same
        return  (this.number==other.number && this.n.equals(other.n) && …) ;
    }
}
```



© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

public boolean equals(Object obj){…

- Equals for Course class
  - identity based on

```
private String courseCode;        (but not on lecturer, room, timetable, …)
private int year;
private String trimester;
```

```
public boolean equals(Object obj){
    if (this==obj)      { return true; }
    if (obj == null)    { return false; }
    if (! obj instanceof Course )       { return false; }
    Course other = (Course) obj;
    return ( this.courseCode.equals(other.courseCode)  &&
     this.year==other.year  &&
     this.trimester.equals(other.trimester) ) ;
}
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Computing Hash Codes

"Wish list" for a hashCode() method:

- Must produce an integer
- Should take account of all components of the object relevant to identity
- Must be consistent with equals()
  - two items that are equal must have the same hash value
- Should distribute the hash codes evenly through the range
  - minimises collisions
- Should be fast to compute

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

A Simple Hash Function for Strings

- We could add up the ascii codes of all the characters:

```
private int hashCode(String value) {
    int hash = 0;
    for (int i = 0; i < value.length(); i++)
        hash += value.charAt(i);
    return hash;
}
```

Why is this not very good?

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Example: Hashing course codes

418 ← DEAF101

419 ← DEAF102 DEAF201

⋮

429 ← BBSC201 MDIA101

430 ← ECHI410 MDIA102 MDIA201

431 ← ECHI303 JAPA111 JAPA201 MDIA202 MDIA220 MDIA301

432 ← ARCH101 ASIA101 BBSC231 BBSC303 BBSC321 CHEM201 ECHI403 ECHI412 JAPA112 JAPA211 JAPA301 MDIA203 MDIA302 MDIA320

⋮

450 ← ANTH412 ARCH389 ARTH111 BIOL228 BIOL327 BIOL372 CHEM489 COML304 COML403 COML421 COMP102 COMP201 CRIM313 CRIM421 DESN215 DESN233 ECON328 ECON409 ECON418 ECON508 EDUC449 EDUC458 EDUC548 EDUC557 ENGL228 ENGL408 ENGL426 ENGL435 ENGL444 ENGL453 FREN124 FREN331 FREN403 FREN412 GEOL362 GEOL407 GERM214 GERM403 GERM412 INFO213 INFO312 INFO402 ITAL206 ITAL215 LALS501 LATI404 LING224 LING323 LING404 MAOR102 MARK304 MARK403 MATH206 MATH314 MATH323 MATH431 MOFI403 PHIL104 PHIL203 PHIL302 PHIL320 PHIL401 PHIL410 RELI321 RELI411 SAMO101

⋮

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei

Better Hash Functions

- Make the contribution of each component depend on its position:

```
public class CourseOffering{
    private final String courseCode;
    private final int year;
    private final char trimester;
    private …    // other fields for timetable, coordinator, …..

    /** hash code depends on the course code, the year, and the trimester */
    public int hashCode() {
        int prime = 104417;
        int hash = year;
        for (int i = 0; i < courseCode.length(); i++)
            hash = hash * prime + courseCode.charAt(i);
        hash = hash * prime  + trimester;
        return hash;
    }
    …
```

© Peter Andreae, Karsten Lundqvist, and Mohammad Nekooei