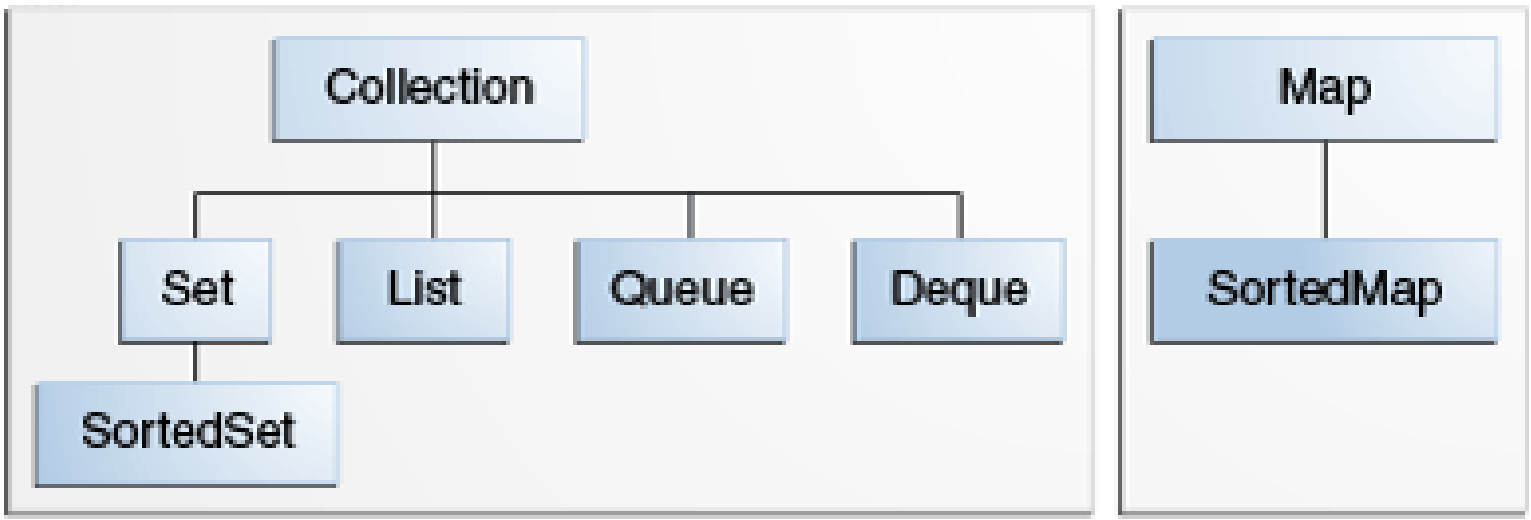
Introduction to Object-Oriented Programming

Hashed Collections

Christopher Simpkins
chris.simpkins@gatech.edu

The Collections Framework



- A *collection* is an object that represents a group of objects.
- The collections framework allows different kinds of collections to be dealt with in an implementation-independent manner.

Well-Behaved Elements

Collections library provides many useful implementations and algorithms. To make maximum use of these collections, your classes should

- Implement `Comparable<T>` so that collections whose elements are instances of your classes can be passed to algorithms that rely on the `Comparable<T>` interface,
- override `equals` so that collections whose elements are instances of your classes can be queried for element membership, and
- override `hashCode` so that hash-based collections will work properly with instances of your classes.

Today we'll learn in more detail why you should override `hashCode` any time you override `equals`.

The `equals` Method and Collections

- A class whose instances will be stored in a collection must have a properly implemented `equals` method.
- The `contains` method in collections uses the `equals` method in the stored objects.
- The default implementation of `equals` (object identity - true only for same object in memory) only rarely gives correct results.
- Note that `hashCode()` also has a defualt implementation that uses the object's memory address. As a rule, whenever you override `equals`, you should also override `hashCode`, which we'll also learn today.

`equals` Method Examples

In this simple class hierarchy, `FoundPerson` has a properly implemented `equals` method and `LostPerson` does not.

```
public class ArrayListEqualsDemo {
    static abstract class Person {
        public String name;
        public Person(String name) { this.name = name; }
    }
    static class LostPerson extends Person {
        public LostPerson(String name) { super(name); }
    }
    static class FoundPerson extends Person {
        public FoundPerson(String name) { super(name); }

        public boolean equals(Object other) {
            if (this == other) return true;
            if (!(other instanceof Person)) return false;
            return ((Person) other).name.equals(this.name);
        }
    }
```

Examine the code in ArrayListEqualsDemo.java to see the consequences.

`equals` and `hashCode`

`java.lang.Object` has another method used by collections:

```
public int hashCode()
```

- The `hashCode` method maps an object to an `int` which can be used to find the object in a kind of data structure known as a hash table.
- Java's hash-based data structures, `HashSet` and `HashMap` use hash tables to store elements and keys.
- The point of a hash code is that it can be computed in constant time, so hashtables allow very fast lookups.
- Every object's `hashCode` method should return a consistent hash code that is not necessarily unique among all objects.

More specifically ...

`hashCode`'s Contract

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

A Correct but Terrible `hashCode()`

```
public int hashCode() {
    return 1;
}
```

This `hashCode` is correct because

- it returns the same value on subsequent invocations,
- `a.hashCode() == b.hashCode()` when `a.equals(b)`, and
- It's legal for `a.hashCode() == b.hashCode()` when `!a.equals(b)`.

However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

A 5-Minute Introduction to Hash Tables

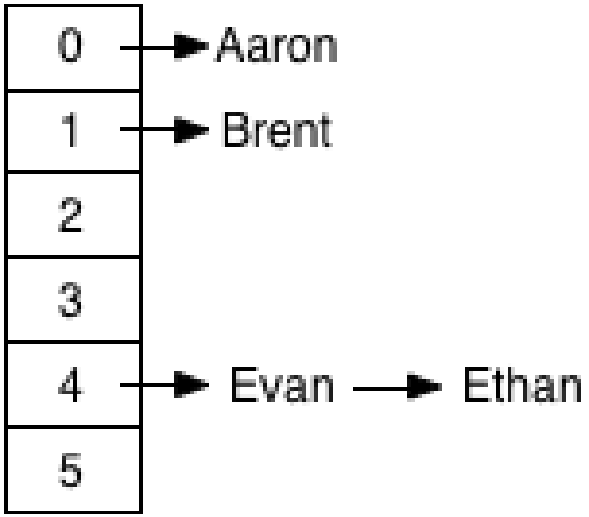
- A hash table stores its elements in "buckets" that are addressed with `int`s.
- An object's `hashCode` determines which bucket the object will be stored in.
- Buckets are accessed very quickly, roughly as quickly as array indexing.
- If each bucket only has one element – because each element has a unique hash code – then every element can be retrieved equally fast.
- When multiple elements have the same `hashCode` (a "hash collision") they go into the same bucket, which stores the elements in a linked list (which has slower access)

An Example Hash Function

Here's a hash function based on the first letter of the name:

```
public class Person {
    private String name;
    public int hashCode() { return name.charAt(0) - 'A'; }
}
```

Then Aaron, Brent, Evan and Ethan would be stored like this:

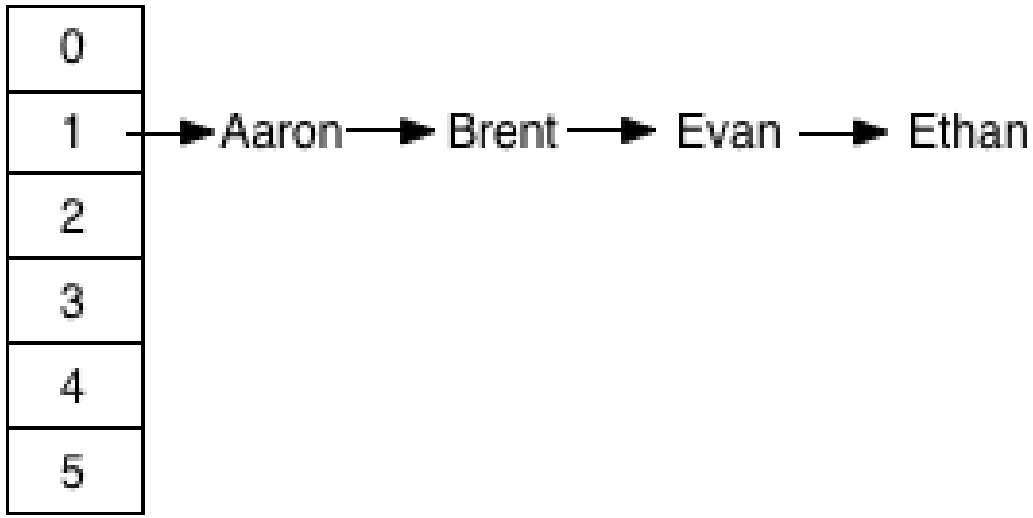


A Legal but Bad `hashCode`

Recall our correct but terrible `hashCode`:

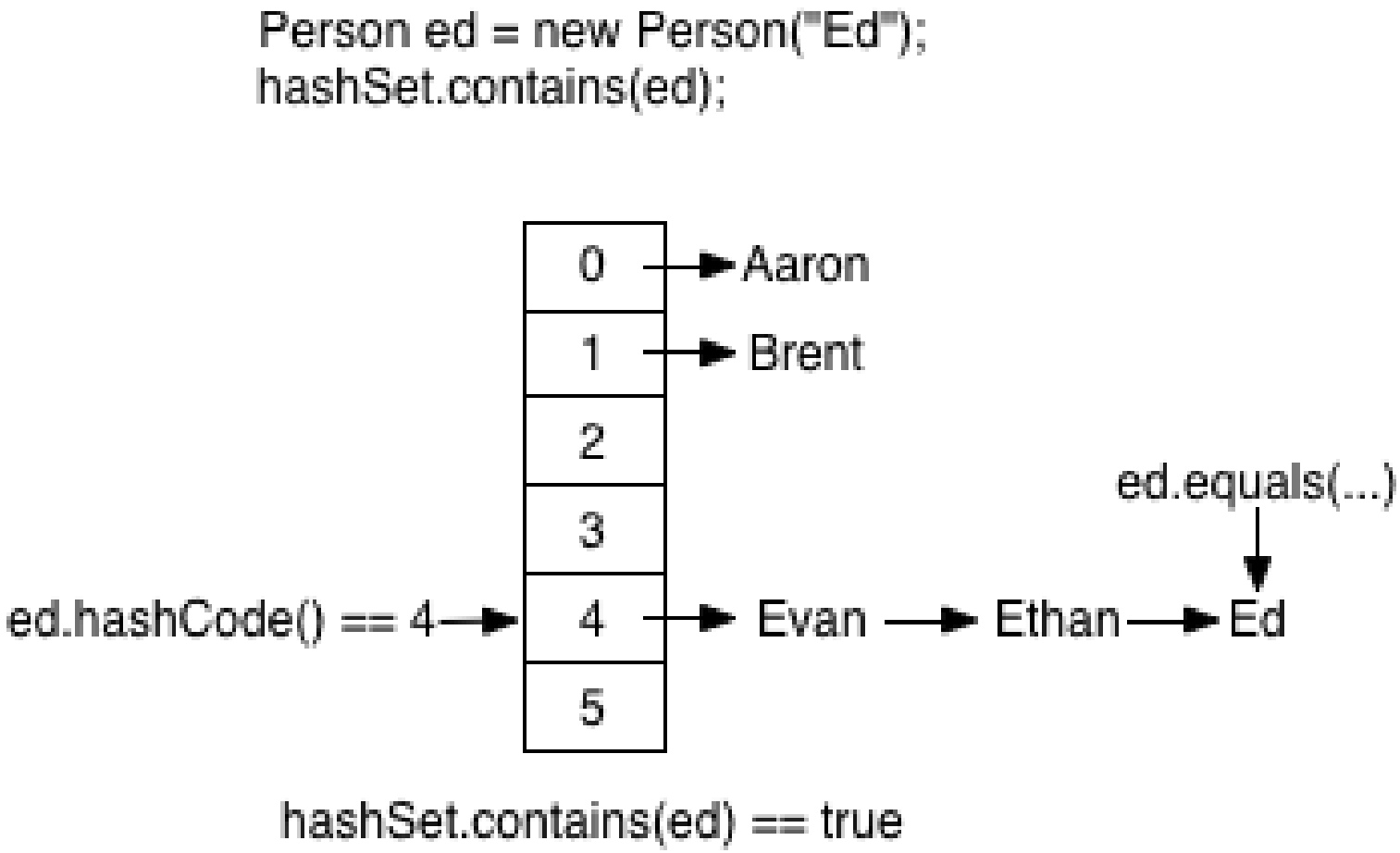
```
public int hashCode() { return 1; }
```

Using this `hashCode` the hash table degenerates to a linked list:



How Items are Found in a Hash-Based Collection

The item's `hashCode` is used to access the right bucket, then its `equals` method is used to match elements in the bucket.



If you override `equals`, you must override `hashCode`!

A Recipe for Implementing `hashCode`[1]

You'll learn hashing in depth in your data structures and algorithms course. For now, here's a recipe to follow:

1. Initialize `result` with a constant non-zero value, e.g., 17
2. For each significant field `f` (i.e., compared in `equals` method), compute an `int` hash code `c` and add it to `31 * result`.
   - For `boolean` fields, `c = (f ? 1 : 0)`
   - For `byte, char, short, int` fields, `c = (int) f`
   - For `long` fields, `c = (int) (f ^ (f >>> 32 ))`
   - For `float` fields, `c = Float.floatToIntBits(f)`
   - For `double` fields,
     ```
     c = (int) (Double.doubleToLongBits(f) ^
          (Double.doubleToLongBits(f) >>> 32))
     ```
     (notice this converts to `long` then uses recipe for `long` fields)
   - For reference fields, if `equals` calls `equals` on the field, `c = f.hashCode()`
   - For array fields, `c = Arrays.hashCode(f)`
3. `return result`

[1] Joshua Bloch, *Effective Java*

An Example `hashCode` Using Recipe[2]

```
class Trooper implements Comparable<Trooper> {

    private String name;
    private boolean mustached;
...
    public boolean equals(Object other) {
        if (null == other) return false;
        if (this == other) return true;
        if (!(other instanceof Trooper)) return false;
        Trooper that = (Trooper) other;
        return this.name.equals(that.name)
                && this.mustached == that.mustached;
    }
    public int hashCode() {
        int result = 17;
        result = 31 * result + name.hashCode();
        result = 31 * result + (mustached ? 1 : 0);
        return result;
    }
}
```

[2] Joshua Bloch, *Effective Java*

Consequences of Failing to Override `hashCode`

```
Set<Trooper> trooperSet = HashSet<>();
// ...
trooperSet.add(new Trooper("Mac", true));

// Mac is in the set, but we don't find him because we didn't
// override hashCode().
System.out.println("\nOops!  Didn't override hashCode():");
System.out.println("trooperSet.contains(new Trooper(\"Mac\", true))="
                   + trooperSet.contains(new Trooper("Mac", true)));
```

prints:

```
Oops!  Didn't override hashCode():
trooperSet.contains(new Trooper("Mac", true))=false
```

Open up SuperTroopers.java and let's fix this!

Closing Thoughts on Collections

- The collections framework uses Java's OOP programming features to acheive generality and consistency.
- Collection classes are very useful - study the Java API docs to become familiar with them.
- In a few weeks we'll implement several basic data structures.
  - Computer scientists need a deep understanding of data structures.
  - Application programmers should almost always use predefined data structures from the standard library.