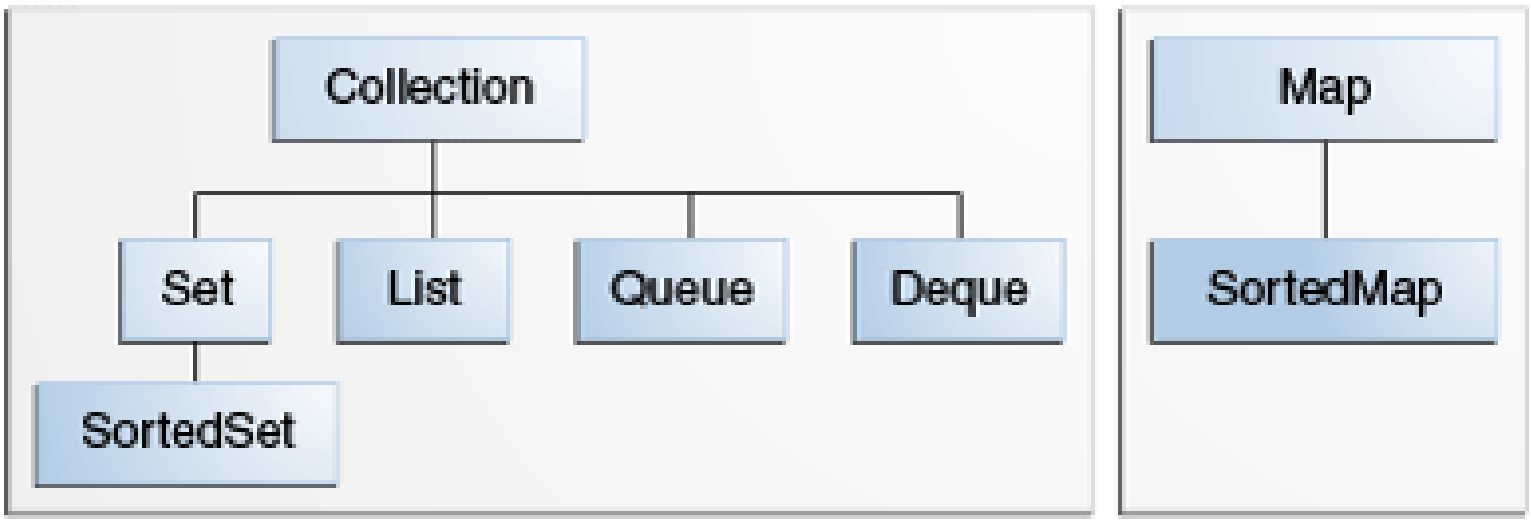
Introduction to Object-Oriented Programming

OOP Case Studies: Collections and JavaFX

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.

Collection Framework Components

The Java collections framework consists of:

- Collection interfaces representing different types of collections (`Set`, `List`, etc)
- General purpose implementations (like `ArrayList` or `HashSet`)
- Abstract implementations to support custom implementations
- Algorithms defined in static utility methods that operate on collections (like `Collections.sort(List<T> list)`)
- Infrastructure interfaces that support collections (like `Iterator`)

`ArrayList` Basics

Create an `ArrayList` with operator `new`:

```
ArrayList tasks = new ArrayList();
```

Add items with `add()`:

```
tasks.add("Eat");
tasks.add("Sleep");
tasks.add("Code");
```

Traverse with for-each loop:

```
for (Object task: tasks) {
    System.out.println(task);
}
```

Note that the for-each loop implicitly uses an iterator.

Using Iterators

Iterators are objects that provide access to the elements in a collection. In Java iterators are represented by the `Iterator` interface, which contains three methods:

- `hasNext()` returns true if the iteration has more elements.
- `next()` returns the next element in the iteration.
- `remove()` removes from the underlying collection the last element returned by the iterator (optional operation).

The most basic and common use of an iterator is to traverse a collection (visit all the elements in a collection):

```
ArrayList tasks = new ArrayList();
// ...
Iterator tasksIter = tasks.iterator();
while (tasksIter.hasNext()) {
    Object task = tasksIter.next();
    System.out.println(task);
}
```

See ArrayListBasics.java for more.

Defining Iterators

```
public class DynamicArray<E> implements Iterable<E> {
    private class DynamicArrayIterator implements Iterator<E> {
        private int cursor = 0;
        public boolean hasNext() {
            return cursor <= DynamicArray.this.lastIndex;
        }
        public E next() {
            cursor++;
            return DynamicArray.this.get(cursor - 1);
        }
        public void remove() { DynamicArray.this.remove(cursor - 1); }
    }
    private Object[] elements;
    private int lastIndex;
    public DynamicArray(int capacity) {
        elements = new Object[capacity]; lastIndex = -1;
    }
    public Iterator iterator() { return new DynamicArrayIterator(); }
```

See DynamicArray.java for examples.

The Iterable Interface

The `Iterable` interface has one abstract method, `iterator`:

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

An instance of a class that implements `Iterable` can be the target of a for-each loop.

```
DynamicArray<String> da = new DynamicArray<>(2);
da.add("Stan");
da.add("Kenny");
da.add("Cartman");
System.out.println("da contents:");
for (String e: da) {
    System.out.println(e);
}
```

Using Generics

Supply a type argument in the angle brackets. Read `ArrayList<String>` as "ArrayList of String"

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("Helluva"); strings.add("Engineer!");
```

If we try to add an object that isn't a `String`, we get a compile error:

```
Integer BULL_DOG = Integer.MIN_VALUE;
strings.add(BULL_DOG); // Won't compile
```

With a typed collection, we get autoboxing on insertion *and* retrieval:

```
ArrayList<Integer> ints = new ArrayList<>();
ints.add(42);
int num = ints.get(0);
```

Notice that we didn't need to supply the type parameter in the creation expression above. Java inferred the type parameter from the declaration. (Note: this only works in Java 7 and above.)

See ArrayListGenericsDemo.java for examples.

Sets

A `Set` is a collection with no duplicate elements (no two elements `e1` and `e2` for which `e1.equals(e2)`) and in no particular order. Given:

```
List<String> nameList = Arrays.asList("Alan", "Ada", "Alan");
Set<String> nameSet = new HashSet<>(nameList);
System.out.println("nameSet: " + nameSet);
```

will print:

```
nameSet: [Alan, Ada]
```

Maps

A `Map<K, V>` object maps keys of type `K` to values of type `V`. The code:

```
Map<String, String> capitals = new HashMap<>();
capitals.put("Georgia", "Atlanta");
capitals.put("Alabama", "Montgomery");
capitals.put("Florida", "Tallahassee");
for (String state: capitals.keySet()) {
    System.out.println("Capital of " + state + " is "
                       + capitals.get(state));
}
```

prints:

```
Capital of Georgia is Atlanta
Capital of Florida is Tallahassee
Capital of Alabama is Montgomery
```

Note that the order of the keys differs from the order in which we added them. The keys of a map are a `Set`, so there can be no duplicates and order is not guaranteed. If you `put` a new value with the same key as an entry already in the map, that entry is overwritten with the new one.

Using `Collections.sort(List<T> list)`

The collections framework includes algorithms that operate on collections implemented as static methods of the `Collections` class. A good example is the `sort` method:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

- `sort` uses the "natural ordering" of the list, that is, the ordering defined by `Comparable`.
- `<? super T>` is a *type bound*. It means "some superclass of `T`."
- The `<T extends Comparable<? super T>>` means that the element type `T` or some superclass of `T` must implement `Comparable`.

See SortTroopers.java for examples.

Can we `Collections.sort(List<T> list)`?

Given the `Collections` static method:

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

And the classes:

```
public class Person implements Comparable<Person>
public class GtStudent extends Person { ... }
```

Can we sort a `List<GtStudent>`?

Type checker "proves" that a type argument satisfies a type specification. Prove by substituting without causing contradictions:

`[GtStudent/T, Person/?]<T extends Comparable<? super T»`

⇒ `<GtPerson extends Comparable<Person super GtStudent>`

We can sort a `List<GtStudent>` becuase

- `GtStudent extends Person`,
- `Person implements Comparable<Person>` and
- `Person` is a supertype of `GtStudent`

Anonymous Inner Classes

We can subclass `Comparator` and make an instance of the subclass at the same time using an *anonymous inner class*. Here's a mustache comparator as an inner class:

```
Collections.sort(troopers, new Comparator<Trooper>() {
    public int compare(Trooper a, Trooper b) {
        if (a.hasMustache() && !b.hasMustache()) {
            return 1;
        } else if (b.hasMustache() && !a.hasMustache()) {
            return -1;
        } else {
            return a.getName().compareTo(b.getName());
        }
    }
});
```

The general syntax for defining an anonymous inner class is

$$\text{new } \textit{SuperType} < \textit{TypeArgument} > () \{\textit{class\_body}\}$$

Functional Interfaces

Any interface with a single abstract method is a functional interface. For example, `Comparator` is a functional interface:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

As in the previous examples, we only need to implement the single abstract method `compare` to make an instantiable class that implements `Comparator`.

Note that there's an optional `@FunctionalInterface` annotation that is similar to the `@Override` annotation. Tagging an interface as a `@FunctionalInterface` prompts the compiler to check that the interface indeed contains a single abstract method and includes a statement in the interface's Javadoc that the interface is a functional interface.

Lambda Expressions

A *lambda expression* is a syntactic shortcut for defining the single abstract method of a funtional interface and instantiating an anonymous class that implements the interface. The general syntax is

$$(T_1\ p_1, ..., T_n\ p_n) \text{ -> } \{method\_body\}$$

Where

- $T_1, ..., T_n$ are types and
- $p_1, ..., p_n$ are parameter names

just like in method definitions.

If *method_body* is a single expression, the curly braces can be omitted. Types in parameter list can also be ommitted where they can be inferred.

MustacheComparator as a Lambda Expression

Here's our mustache comparator from SortTroopers.java as a lambda expression:

```
Collections.sort(troopers, (Trooper a, Trooper b) -> {
    if (a.hasMustache() && !b.hasMustache()) {
        return 1;
    } else if (b.hasMustache() && !a.hasMustache()) {
        return -1;
    } else {
        return a.getName().compareTo(b.getName());
    }
});
```

- Because `Collections.sort(List<T> l, Comparator<T> c)` takes a `Comparator<T>`, we way that `Comparator<T>` is the *target type* of the lambda expression passed to the `sort` method.
- The lambda expression creates an instance of an anonymous class that implements `Comparator<Trooper>` and passes this instance to `sort`

Target Types

```
static interface Bar {
    int compare(Trooper a, Trooper b);
}
static void foo(Bar b) { ... }
```

Given the `Bar` interface, the call:

```
foo((Trooper a, Trooper b) -> {
    if (a.hasMustache() && !b.hasMustache()) {
        return 1;
    } else if (b.hasMustache() && !a.hasMustache()) {
        return -1;
    } else {
        return a.getName().compareTo(b.getName());
    }
});
```

creates an instance of the `Bar` interface using the same lambda expression.

- The type of object instantiated by a lambda expression is determined by the *target type* of the call in which the lambda expression appears.

Streams and Pipelines

A stream is a sequence of elements.

- Unlike a collection, it is not a data structure that stores elements.
- Unlike an iterator, streams do not allow modification of the underlying source

A collection provides a source for a pipeline, which processes a stream derived from the source.

A pipleline carries values from a source to a sink.
A pipeline contains:

- A source: This could be a collection, an array, a generator function, or an I/O channel.
- Zero or more intermediate operations. An intermediate operation, such as filter, produces a new stream
- A terminal operation. A terminal operation, such as forEach, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of forEach, no value at

Method References

Three kinds of method references:

- *object*::*instanceMethod* - like `x -> object.instanceMethod(x)`
- *Class*::*staticMethod* - like `x -> Class.staticMethod(x)`

```
someList.forEach(System.out::println);
```

- *Class*::*instanceMethod* - like `(x, y) -> x.instanceMethod(y)`

```
Comparator<Trooper> byName =
    Comparator.comparing(Trooper::getName);
```

Stream Example: How Many Mustaches?

Consider this simple example from SortTroopers.java:

```
long mustaches =
    troopers.stream().filter(Trooper::hasMustache).count();
System.out.println("Mustaches: " + mustaches);
```

- `troopers.stream()` is the *source*
- `.filter(Trooper::hasMustache)` is an *intermediate operation*
- `.count()` is the *terminal operation*, sometimes called a *sink*

The terminal operation yields a new value which results from applying all the intermediate operations and finally the terminal operation to the source.
See StreamTroopers.java for examples.

Stream Example: Long Words

Given:

```
List<String> words = Arrays.asList("Hello", "World", "Welcome", "To",
    "Java", "8");
```

Write a single statement that assigns to `avg` the average word length:

```
double avg = words.stream()
    .map(String::length)
    .reduce(0, (a, b) -> a + b) / (0.0 + words.size());
```

Using `words` and `avg`, write a single statement that collects a list of all words in `words` that are longer than `avg`, and assigns this list to a properly typed `List<T>`:

```
List<String> longWords = words.stream()
    .filter(word -> word.length() > avg)
    .collect(Collectors.toList());
```

See this and other stream examples in Streams.java

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`

A Recipe for Implementing `equals(Object)`

Obeying the general contract of `equals(Object)` is easier if you follow these steps.

1. Ensure the other object is not null.
2. Check for reference equality with == (are we comparing to self?).
3. Check that the other object is an `instanceof` this object's class.
4. Cast the other object to this's type (guaranteed to work after instanceof test)
5. Check that each "significant" field in the other object `equals(Object)` the corresponding field in this object.

After seeing an example applicaiton of this recipe we'll motivate the proper implementation of `equals(Object)` methods by introducing our first collection class, `ArrayList`.

An Example `equals(Object)` Method

Assume we have a `Person` class with a single `name` field.

1. Ensure the other object is not null.
2. Check for reference equality with == (are we comparing to self?).
3. Check that the other object is an `instanceof` this object's class.
4. Cast the other object to this's type (guaranteed to work after instanceof test)
5. Check that each "significant" field in the other object `equals(Object)` the corresponding field in this object.

Applying the recipe:

```
        public boolean equals(Object other) {
1:          if (null == other) { return false; }
2:          if (this == other) { return true; }
3:          if (!(other instanceof Person)) { return false; }
4:          Person that = (Person) other;
5:          return this.name.equals(that.name);
        }
```

Conequences of Failing to Override `equals(Object)`

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

```
abstract static 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);
    }
}
```

hashCode

Hash-based implementations, `HashSet` and `HashMap`, store and retrieve elements or keys using the `hashCode` method from `java.lang.Object`:

```
public int hashCode()
```

- The `hashCode` method maps an object to an `int` which can be used to find the object in a data structure called a `hashtable`.
- 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.

Bottom line: 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*

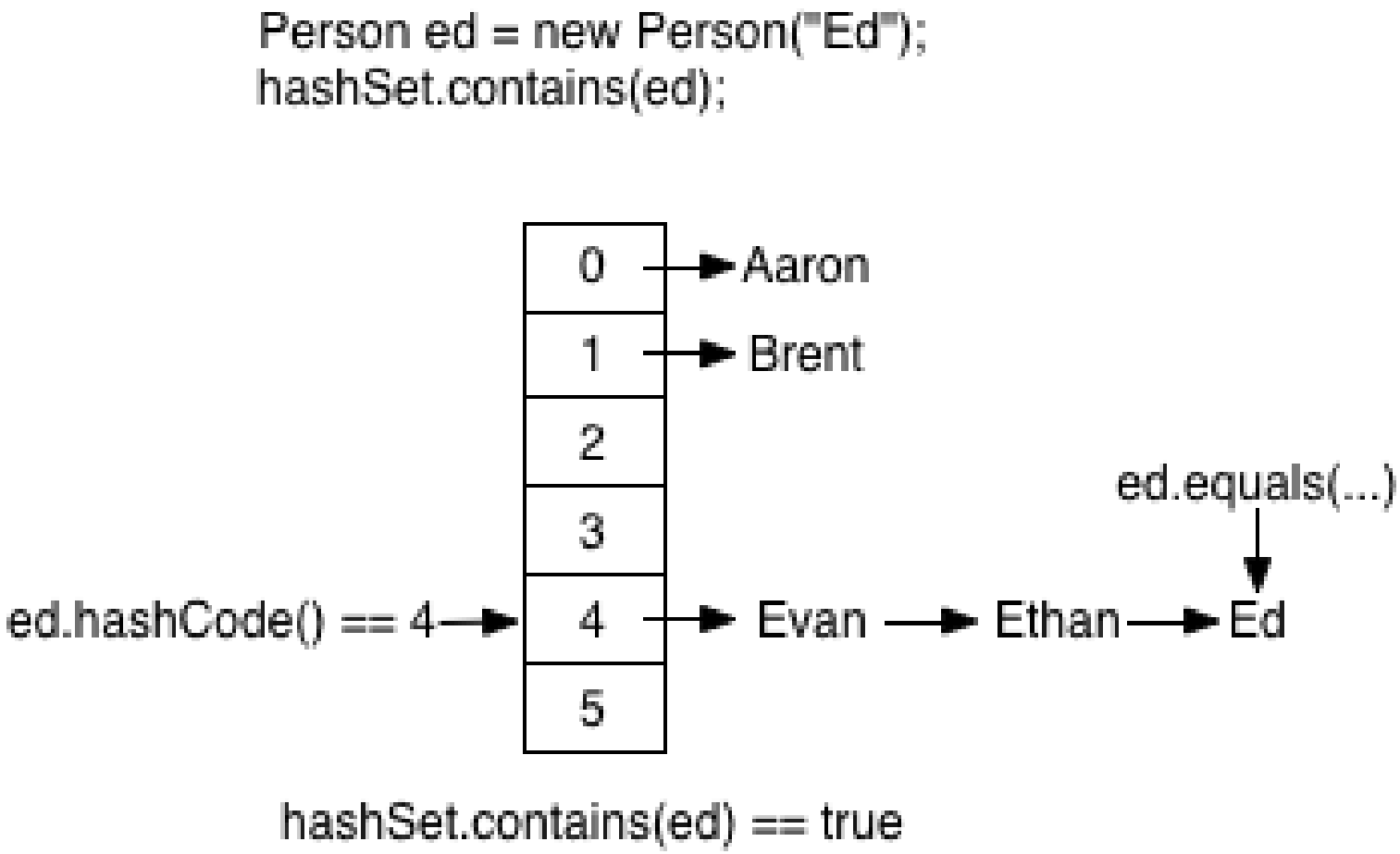
A Simpler Recipe for Implementing `hashCode`

The basic idea is to add some `int` value for each significant field. Joshua Bloch's recipe works well for Java's collections, but a crude approximation is also fine:

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 all numeric primitives, perform an explicit conversion to `int`,** `c = (int) f`
   - For reference fields, if `equals` calls `equals` on the field, `c = f.hashCode()`
   - For array fields, `c = Arrays.hashCode(f)`
3. `return result`

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`!

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
```