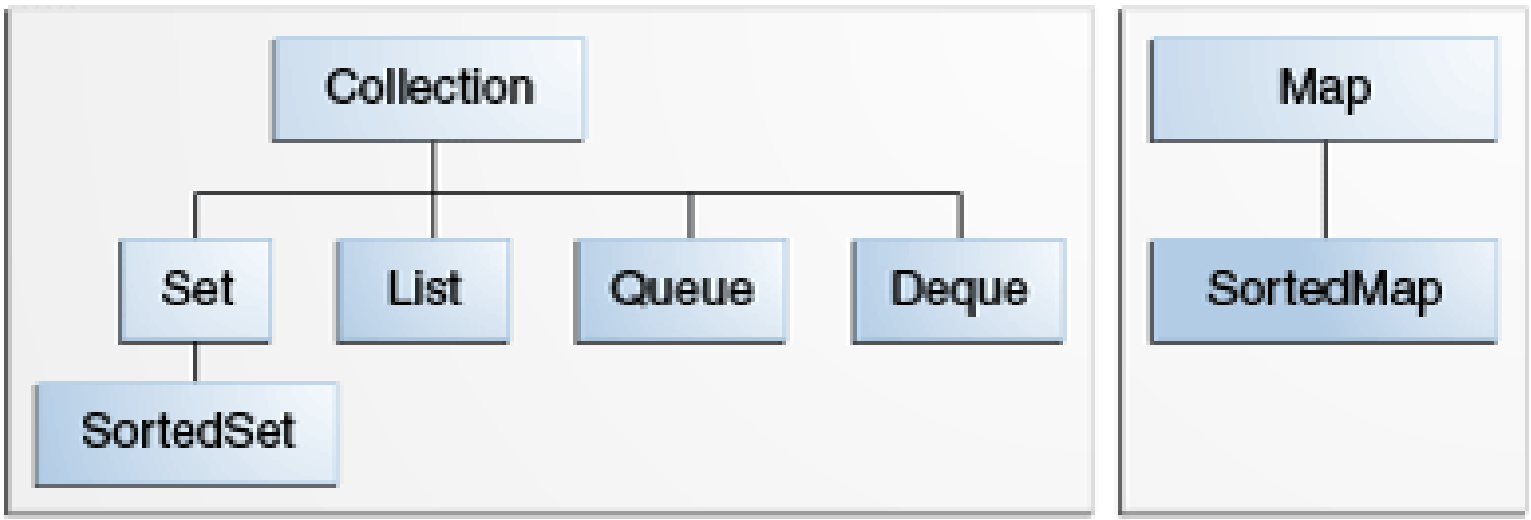
Java Collections (Part 1 of 3)

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 (sets, lists, etc)
- General purpose implementations (like `ArrayList` or `HashSet`)
- Absract 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`)

Today we'll learn a few basic concepts, then tour the collections library.

The `Collection` Interface

`Collection` is the root interface of the collections framework, declaring basic operations such as:

- `add(E e)` to add elements to the collection
- `contains(Object key)` to determine whether the collection contains `key`
- `isEmpty()` to test the collection for emptiness
- `iterator()` to get an interator over the elements of the collection
- `remove(Object o)` to remove a single instance of `o` from the collection, if present
- `size()` to find out the number of elements in the collection

None of the collection implementations in the Java library implement `Collection` directly. Instead they implement `List` or `Set`.

Lists and `ArrayList`

The `List` interface represents ordered collections, or *sequences*. `List` adds

- methods for positional (indexed) access to elements (`get(int index)`, `indexOf(Object o)`, `remove(int index)`, `set(int index, E element)`),
- a special iterator, `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides; and methods to obtain a `ListIterator`
- a `subList(int fromIndex, int toIndex)` that returns a view of a portion of the list.

`ArralyList` and `LinkedList` are the two basic `List` implementations provided in the Java standard library.[1]

[1] `Vector` also implements `List` and can be thought of as a synchronized version of `ArrayList`. You don't need `Vector` if you're not writing multithreaded code. Using `Vector` in single-threaded code will decrease performance.

`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.

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.

Generics

Did you notice the warning when we compile `ArrayListBasics.java`?

```
$ javac ArrayListBasics.java
Note: ArrayListBasics.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Java issues this warning because `ArrayList` (and the other collecttion classes in the Java library) is a *parameterized type* and we used `ArrayList` without a type parameter. The full class name is `ArrayList<E>`.

- `E` is a *type parameter*, which can be any class name (not a primitive type).
- `ArrayList<E>` is a *parameterized type*
- `E` tells the compiler which types are stored in the collection.

So the compiler is warning us that we're not using the type parameter and thus missing out on static type-checking.

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 more.

Primitives in Collections

`ArrayList`s can only hold reference types. So you must use wrapper classes for primitives:

```
ArrayList ints = new ArrayList();
ints.add(new Integer(42));
```

Java auto-boxes primitives when adding to a collection:

```
ints.add(99);
```

But auto-unboxing can't be done when retrieving from an untyped collection:

```
int num = ints.get(0); // won't compile
```

The old way to handle this with untyped collections is to cast it:

```
int num = (Integer) ints.get(0); // auto-unboxing on assignment to int
```

See ArrayListPrimitivesDemo.java for more.

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>` is an object that 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.

Programming Exercise

Write a class called `WordCount`.

- The constructor should take a `String` file name.
- `WordCount` should have an instance variable `wordCounts` which is a `Map` from `String` to `int`, where each `String` key is a word that occurs in the file supplied to the constructor, and the corresponding `int` is the number of times the word appears in the file.

Extra: normalize the word counts to $[0, 1]$ so that the word counts represent the probability that a randomly chosen word from the file is a given word. For normalized word counts, what will be the type of the value in the map?