Introduction to Object-Oriented Programming

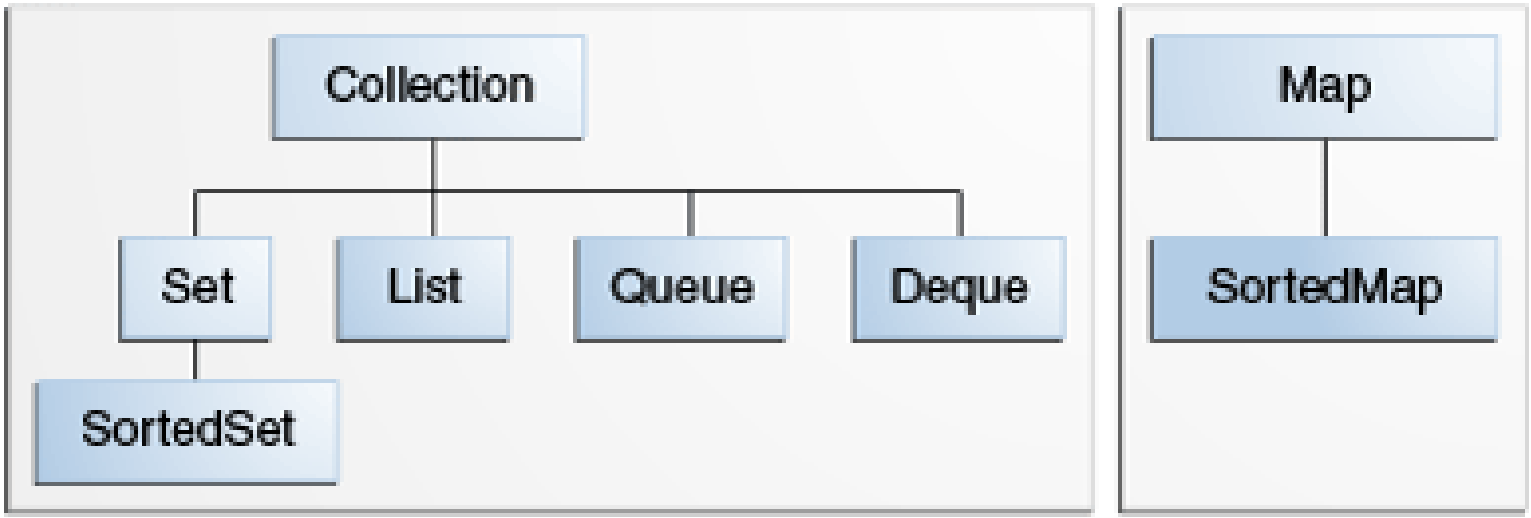
ArrayList

Christopher Simpkins
chris.simpkins@gatech.edu

Java Collections and `java.util.ArrayList`

- The Java collections hierarchy
- Arrays and `ArrayList`
- ArrayList basics
- Primitives in Collections
- Generics
- The `equals` Method and Collections

Java Collections



`ArralyList` and `LinkedList` are the two basic `List` implementations provided in the Java standard library.[1] The concepts we'll learn for `ArrayList` apply to all of Java's collection classes.

[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.

Arrays and `ArrayList`

- Arrays are fixed-size collections of any data types, including primitives
- `ArrayList`s are dynamically-allocated (i.e., automatically resized) collections of reference types (not primitives - but we'll talk about autoboxing).
- `ArrayList`s use arrays internally, but this isn't important to know for basic use.

`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 objects 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.

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
```

We'll see a better way to handle this with generics.
See ArrayListPrimitivesDemo.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.

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`[2].

[2] `hashcode()` is used in objects that are keys in `Map`s. You'll learn about `Map`s later in the course.

`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

Closing Thoughts on Collections and `ArrayList`

- Collection classes are very useful - study the Java API docs to become familiar with them.
- The concepts we just learned about `ArrayList` apply to all collections.
- 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.
- For now, knowing how to use Java collections is an important skill for any Java programmer, and collections are used extensively in Swing.