Java Programming 2
Lecture #10

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
23 October 2019

"Taste the Rainbow" by Christopher Porter
https://flic.kr/p/c86atW

Outline

Java Collections framework recap

More on the `ArrayList` class

Iterating over Collections

Generic types

Other useful classes: `HashMap, Set`

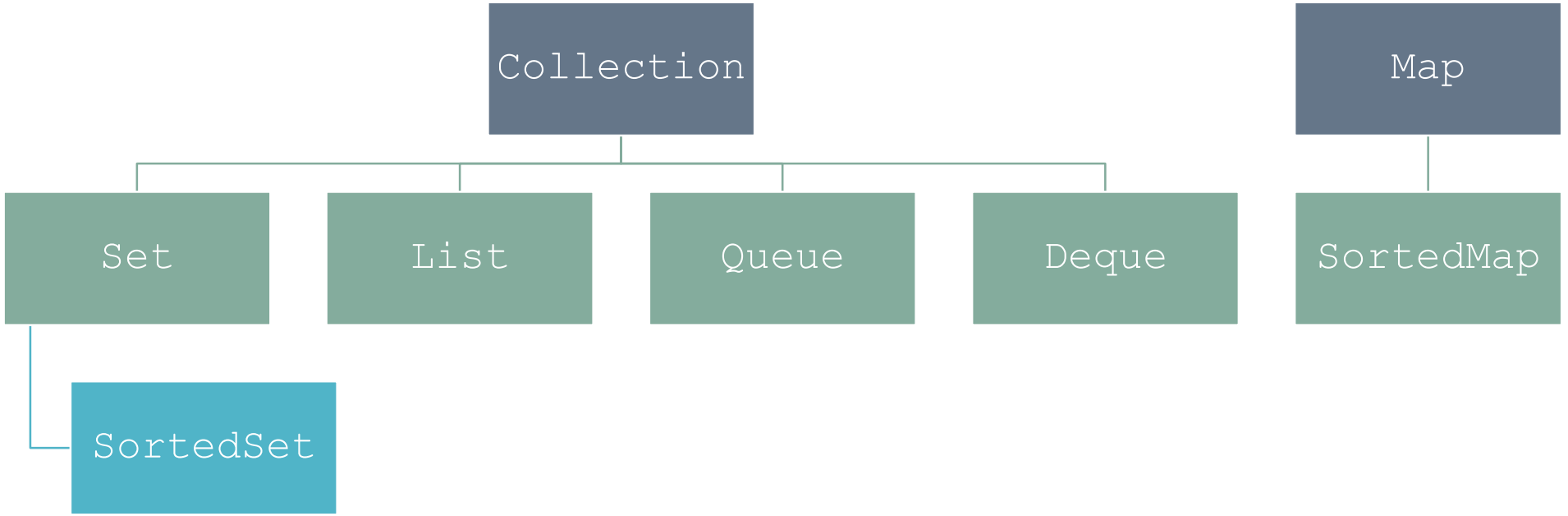
Feedback on Java/Python quiz

Mid-course survey

Structure of Collections framework

Base class:* `java.util.Collection`

Methods: `add(), remove(), contains(), size(), toArray()`



** Actually, everything in this picture is technically an interface – we will discuss the details next week.*

java.util.ArrayList

A Collections class (specifically, a `List`) that implements **variable-length** arrays

More flexible than built-in arrays, but less efficient

Acts as a wrapper around an underlying array that grows and shrinks dynamically

`ArrayList` is a **class** – so elements are added and removed by **methods**
(Not by built-in Java syntax as with normal arrays)

It has a **capacity** (size of internal array) and a **size** (number of elements in the list)

Capacity is increased when necessary – purely internal

Size is increased/decreased as elements are added and removed, and checked for operations

In general: IndexOutOfBoundsException if `(index < 0 || index >= size())`

`List` vs `ArrayList`?

`List` is the **high-level Collection type** (the interface – think of it as an abstract class for now)

- Specifies methods including **add**, **clear**, **isEmpty**, **remove**, **set**, …

`ArrayList` is the **specific type of List**

- Provides a concrete implementation
- Additional methods related to capacity

When to use which?

- Use `ArrayList` …
  - *When initialising a new variable*
  - *If you want to manipulate capacity*
- Use `List` all other times – allows implementations to be swapped cleanly

Converting to and from normal arrays

```
// Convert a List to an array
List<String> strList = new ArrayList<>();
String[] strArray =
         strList.toArray(new String[strList.size()]);

// Convert an array to a List
String[] strings = { “each”, “peach”, “pear”, “plum” };
List<String> stringList = Arrays.asList (strings);
```

Bonus: ArrayList has `toString()`!

```
String[] words = { “each”, “peach”, “pear”, “plum” };
System.out.println (words);
// Prints “[Ljava.lang.String;@659e0bfd”

List<String> wordList = Arrays.asList (words);
System.out.println (wordList);
// Prints “[each, peach, pear, plum]”
```

Iterating over ArrayLists – same as arrays

```
for (int i = 0; i < words.size(); i++) {
    String s = words.get (i);
    System.out.print (s + " ");
}
// or …
for (String s : words) {
    System.out.print (s + " ");
}
```

Generic types


```
List<String> strList = new ArrayList<>();
```

Collection classes are **type-parameterised**

- The type specified in angle brackets after the name specifies the type of the elements stored in that Collection
- If you don't specify any type, then it will use `java.lang.Object`

(Polymorphism: subclasses of specified type will also be accepted)

Generic types were added to Java in Java 1.5 (2004)

Why use generic types?

Compile-time error checking

```
List<String> strList = new ArrayList<>();
strList.add ("foo");
strList.add (new java.util.Scanner()); // fail
```

Iteration can be much cleaner (especially with new-style iteration)

```
for (String s : words) {
    System.out.println (s.toLowerCase());
}
```

```
for (int i = 0; i < words.size(); i++) {
    String s = (String)words.get(i);
    System.out.println (s.toLowerCase());
}
```

Primitive types and generics

The <*type*> generic parameter needs to be a **class**

Primitive types will not work!

~~`List<int> intList;`~~

Solution: Use **wrapper** classes (`int`/`Integer`, `long`/`Long`, etc.)

```
List<Integer> intList = new ArrayList<>();
```

But you don't want to have to do this all the time …

```
Integer i2 = new Integer (i);
intList.add (i2);
int value = intList.get(5).intValue();
```

Boxing and unboxing

Good news: Java **automatically** converts between wrapper classes and primitive types

(Also since Java 1.5)

```
List<Integer> intList = new ArrayList<>();
intList.add (5);
intList.add (10);
int value = intList.get (0);
Integer value2 = intList.get(1) * 100;
```

Sample (Array)List code: Fibonacci sequence

```
List<Integer> fibonacci (int limit, int sizeLimit) {
        List<Integer> nums = new ArrayList<>();
        nums.add(1);
        nums.add(1);
        int i = 2;
        int fib = 1;
        while (fib < limit && nums.size() < sizeLimit) {
                fib = nums.get(i-1) + nums.get(i-2);
                nums.add(fib);
                i++;
        }
        return nums;
}
```

Sets

Interface: `java.util.Set`

- Concrete implementations: `HashSet, TreeSet, LinkedHashSet`

Differences to List

- Cannot contain duplicate elements
  - *add() method enforces this – returns true/false indicating if element was already in set*
- Two sets are equal if they contain the same elements, regardless of implementation

Using a Set to find unique values

```
Collection<String> findDistinct(Collection<String> input)
{
      Set<String> distinct = new HashSet<> (input);
      return distinct;
}
```

Maps

Interface: `java.util.Map`

- Concrete implementations: `HashMap, TreeMap, LinkedHashMap`

Provides a mapping from keys to values

- Cannot contain duplicate keys
- Each key maps to exactly one value

Useful methods:

- `get(key)` – return the value associated with a key (null if no value)
- `put(key, value)` – set the new value associated with that key

Using a Map to count word frequency

```
Map<String, Integer> countWords(Collection<String> input) {
	Map<String, Integer> result = new HashMap<>();
	for (String word : input) {
		Integer value = result.get(word);
		if (value == null) {
			value = 0;
		}
		result.put(word, value+1);
	}
	return result;
}
```

Java/Python feedback

Mid-course survey

https://tinyurl.com/jp2-2019-mid-course

Next time

Friday: going over Lab 4; introduction to Lab 5

Monday:

- Interfaces
- Javadoc