Java Programming 2
Generics; other Collections

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

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

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;
}
```