SWEN20003
Object Oriented Software Development
Workshop 8

Eleanor McMurtry

Semester 2, 2020

Workshop

This week is all about **generics**.

1. Generics have a **parameterised type**, usually written as a capital letter (e.g. `<T>`). This is one kind of **polymorphism**.
2. Classes and interfaces can be generic, such as `Comparable<T>`.
3. Methods can also be generic without their containing class be generic.
4. The class `ArrayList<T>` is a generic automatically-resizing array. We usually use this class instead of arrays in practice.
5. The class `HashMap<K, V>` is a **dictionary** mapping keys of type `K` to values of type `V`. Internally, it works like a **Hash table** (the details of this will not be assessed).

Questions

1. Write an immutable class `Pair` that accepts two type parameters `T` and `U`, and simply stores two objects (one of type `T` and one of type `U`). The class should take in the two objects in its constructor.

   Solution:

```
public class Pair<T, U> {
    public final T first;
    public final U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }
}
```

2. The `Number` abstract class is part of the Java standard library and is the superclass of all numeric primitive wrapper classes such as `Integer`, `Double`, `Long`, e.t.c. Using this knowledge, create a new class `Rectangle` that contains two instances of `Pair`, `topLeft` and `bottomRight` (with both values in the pairs the same type), but allows only subclasses of `Number` to be stored.
   Hint: use a bounded type parameter (`<T extends ...>`).

   Solution:

```
public class Rectangle<T extends Number> {
    public final Pair<T, T> topLeft;
    public final Pair<T, T> bottomRight;

    public Rectangle(T top, T left, T bottom, T right) {
        this.topLeft = new Pair<>(top, left);
        this.bottomRight = new Pair<>(bottom, right);
    }
}
```

3. Implement a method that processes an `ArrayList<String>`, and returns a single `String`, which is a comma-separated list of all the items in the input that contain **only** a single word.

   Solution:

```
public static String getSingleWords(ArrayList<String> strings) {
    String output = "";
    for (String s : strings) {
        if (s.split(" ").length == 1) {
            output += "," + s;
        }
    }
    return output.substring(1);
}
```

4. Implement a method that takes a string as an argument and returns a `HashMap<Character, Integer>` containing the number of times each character in the string appears.

   Solution:

```
public static HashMap<Character, Integer> frequencyCount(String str) {
    HashMap<Character, Integer> result = new HashMap<>();

    for (int i = 0; i < str.length(); ++i) {
        char c = str.charAt(i);
        result.putIfAbsent(c, 0);
        result.put(c, result.get(c) + 1);
    }

    return result;
}
```

5. A *tree* is a data structure that is made up of **nodes**. At the most basic level, each node holds a reference to its children, and a value. A *binary tree* is a tree that has at most two children per node, conventionally referred to as "left" and "right".

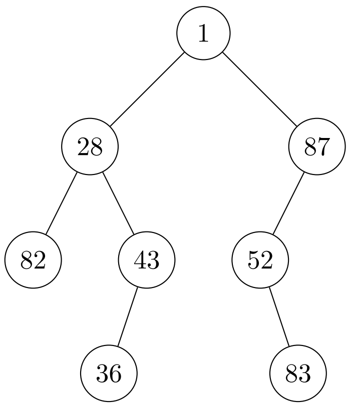

Figure 1: A simple binary tree

   Implement a generic class that can represent a binary tree. (Note that you don't have to keep the nodes in sorted order for this problem—but it would be a good extension task!)

   Solution:

```
public class Node<T> {
    public final T value;

    public final Node<T> left;
    public final Node<T> right;

    public Node(T value, Node<T> left, Node<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

}

public class BinaryTree<T> {
    public final Node<T> root;

    public BinaryTree(Node<T> root) {
        this.root = root;
    }
}
```

6. Write a custom generic class `CycleList<T>` that is like a regular list, except it can be *cycled*: it should define a method `T next()` that returns the next item in the list, starting at the 0th. When the end of the list is reached, it should return to the beginning of the list. You should also define:

   (a) `void add(T value)`

   (b) `boolean contains(T value)`

   (c) `void remove(T value)`

   (d) `void addAll(Collection<T> collection)`

   Solution:

```
class CycleList<T> {
    private List<T> items = new ArrayList<>();
    private int iterator = 0;

    public T next() {
        T item = items.get(iterator++);
        iterator = iterator % items.size();
        return item;
    }

    public void add(T value) {
        items.add(value);
    }

    public boolean contains(T value) {
        return items.contains(value);
    }

    public void addAll(Collection<T> collection) {
        items.addAll(collection);
    }

    public void remove(T item) {
        items.remove(item);
    }
}
```

7. Write a class `SortedCycleList<T extends Comparable<T>>` that acts like a `CycleList<T>`, except the `next()` method cycles through items in sorted order.

   Solution:

```
class SortedCycleList<T extends Comparable<T>> {
    private List<T> items = new ArrayList<>();
    private int iterator = 0;

    public T next() {
        T item = items.get(iterator++);
        iterator = iterator % items.size();
        return item;
    }

    public void add(T value) {
        items.add(value);
        Collections.sort(items);
    }

    public void addAll(Collection<? extends T> collection) {
        items.addAll(collection);
        Collections.sort(items);
    }

    public void remove(T item) {
        items.remove(item);
    }
}
```