Lecture 25 - 4/16/2023

The central focus of this lecture is recursion. Recursion is a method for solving computational problems that involves calling the same method that you are currently in with a smaller subcase of the one you were originally working on. A classic example of a recursive algorithm is for dealing with factorials:

```
public static int factorial2(int n){
    if (n<=1){
         return 1;
    }else{
         return n*factorial2(n-1);
    }
}
```

From this method we can see the three laws of recursion that are necessary to write proper recursive methods

The three laws of recursion are:

1. You must always have a base case
2. You must call the method within itself
3. You must make progress towards the base case with each recursive call

Rule 1: You must always have a base case

As we can see our base case is when n <= 1 it is the first thing we check for in our method.

Rule 2: You must call the method within itself

Pretty much the definition of recursion, we can see this happen on the last line of the code snippet above.

Rule 3: You must make progress towards the base case with each recursive call.

With this you kinda have to trust yourself at first, but you should always be making your recursive calls with an argument that gets you closer to the base case!

If you don’t do this you run the risk of getting the following error:

java.lang.StackOverflowError

So please make sure you are approaching the base call with each recursive call!

It is also important to note that any program with a recursive solution also has an iterative solution as well.

Lecture 26 - 4/18/2023

In this lecture, we began by wrapping up our discussion of recursion by introducing mergesort. Which in our case is a recursive sorting algorithm that divides up an array of elements into continuous halves until the bits are small enough to merge back together in sorted order. Here is a simplified version of the mergesort algorithm:

```
public void mergeSort(int arr[], int l, int r){
        if (l < r){
            // Find the middle point
            int m = (l+r)/2;

            // Sort first and second halves
            mergeSort(arr, l, m);
            mergeSort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
}
```

Of course, I am leaving out the implementation of the merge method for formatting purposes and to remain succinct, if you wish to view more about mergesort please do so by following this link

We finished up the lecture by introducing you to Javadoc!

Javadoc is a documentation system that allows us to write documentation just like the normal java api that you can google right now!

We begin javadoc comments with a slash and two asterisk:

```
/**
 *
 */
```

In the comment we can write a description of our method and identify parameters and return values with: @param @return the following is an example of a javadoc comment on a method:

```
/**
 *  Returns some value related to our method
 *
 *  @param a  an int to be provided to the method
 *  @return a double representing some value
 */
public double myMethod(int a){}
```

There are other tags you can use view this javadoc guide finally we can compile our javadoc using the javadoc command:

```
javadoc *.java
```

NOTE AS OF APRIL 16th 2024 Generics has been axed from the curriculum but the below discussion is still included for completeness

This will also be where we formally introduce you to generics or parameterized methods and classes in Java. You have actually already seen the applications of generics before, if you have ever declared an ArrayList you have successfully applied generics. Rather than have an explicit type we will leave it up to the user which type they use. Here is the merge sort algorithm again but using generics:

```
public void <T extends Comparable<T>> mergeSort(T arr[], int l, int r){
        if (l < r){
            int m = (l+r)/2;

            // Sort first and second halves
            mergeSort(arr, l, m);
            mergeSort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
}
```

Nothing has really changed, we are just saying we can run this method on any Type T that is Comparable (makes sense for a sorting or searching algorithm or any algorithm where you are performing comparisons between elements) Anything we wish to be generic we just replace with a generic variable, like T. We can do the same for classes too naturally.

```
public class ThisClassIsGeneric<T> {
```

```
      //whatever you fill your class with
}
```

When we make an instance of a generic class we do so using the angular brackets like with an ArrayList.

```
       ThisClassIsGeneric<Integer> myObject = new ThisClassIsGeneric<>();
```

Of course we are also not limited to a single generic element, we can have multiple. Take the following example which uses two generic elements:

```
public class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
          this.key = key;
          this.value = value;
    }

    public K getKey(){ return key; }

    public V getValue() { return value; }
}
```