OOP Assignment IV

1. **Why do we need generics in java? Explain with an example?**

In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Code that uses generics has many benefits over non-generic code:

Stronger type checks at compile time.

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Elimination of casts.

The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Enabling programmers to implement generic algorithms.

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

2. **What is Bounded wildcards in Generics? Explain with example?**

Bounded and Unbounded wildcards in generics are used to bound any Type.

Type can be upper bounded by using `<? extends T>` where **all Types must be sub-class of T** or lower bounded using `<? super T>` where **all Types required to be super class of T**, here T represent lower bound.

Single `<?>` is called an unbounded wildcard in generic and it can represent any type, similar to Object in Java. For example `List<?>` can represent any List e.g. `List<String>` or `List<Integer>` its provides highest level of flexibility on passing method argument.

For example, consider the following code snippet

```
public static void process (List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

The upper bounded wildcard, `<? extends Foo>`, where `Foo` is any type, matches `Foo` and any subtype of `Foo`. The `process` method can access the list elements as type `Foo`.

Say you want to write a method that puts `Integer` objects into a list. To maximize flexibility, you would like the method to work on `List<Integer>, List<Number>,` and `List<Object>` — anything that can hold `Integer` values.

You can do so by a lower bounded wildcard:

```
public static void operate(List<? super Integer> list)
```

3. **Explain about generic methods and generic constructors with example?**

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

For example:

The `Util` class includes a generic method, `compare`, which compares two `Pair` objects:

```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K,
V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    public Pair (K key, V value) {
          setKey(key);
          setValue(value);
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

4. **What is meant by immutability of String Type object? How to find all rotations of a given String? For example if the string is "ABCD" it will output ABCD, BCDA, CDAB and DABC.**

String type objects are immutable.

For example:

```
String s1 = "Hello";
String s2 = s1;
// s1 and s2 now point at the same string - "Hello"
```

Now, there is nothing we could do to `s1` that would affect the value of `s2`. They refer to the same object - the string `"Hello"` - but that object is immutable and thus cannot be altered.

```
s1 = "Help!";
System.out.println(s2); // still prints "Hello"
```

Here we see the difference between mutating an object, and changing a reference. `s2` still points to the same object as we initially set `s1` to point to. Setting `s1` to `"Help!"` only changes the *reference*, while the `String` object it originally referred to remains unchanged.

```
import java.util.Scanner;
class Rotate {
	public static void main (String [] args) {
		Scanner sc = new Scanner (System.in);
		System.out.print("Enter the string: ");
		String str = sc.nextLine();
		int l = str.length();
		for (int i = 0; i < l; ++i) {
			for (int j = i; j < l; ++j)
				System.out.print(str.charAt(j));
			for (int j = 0; j < i; ++j)
				System.out.print(str.charAt(j));
			System.out.println();
		}
	}
}
```

5. **Explain about following String methods**

   1)indexOf()

   2)regionMatches()

   3)substring()

   4)replace()

   Also Write a program that use all the above methods.

`int indexOf(int ch)`

Returns the index within this string of the first occurrence of the specified character.

`int indexOf(int ch, int fromIndex)`

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

`int indexOf(String str)`

Returns the index within this string of the first occurrence of the specified substring.

`int indexOf(String str, int fromIndex)`

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index

`boolean regionMatches(int toffset, String other, int ooffset, int len)`

Tests if two string regions are equal

`String substring(int beginIndex)`

Returns a new string that is a substring of this string.

`String substring(int beginIndex, int endIndex)`

Returns a new string that is a substring of this string.

`String replace(char oldChar, char newChar)`

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

```
class StringDemo {
    public static void main (String [] args) {
      String s1 = "APPLE";
      String s2 = "ANDROID";
      System.out.println(s1.indexOf('A'));                    // 0
      System.out.println(s2.indexOf('D', 5));                 // 6
      System.out.println(s1.regionMatches(0, s2, 2, 2));      // false
      System.out.println(s2.substring(2));                    // DROID
      System.out.println(s1.substring(1, 3));                 // PP
      System.out.println(s2.replace('O', 'A'));               // ANDRAID
    }
}
```