# Lambdas

First 2 definitions as a reminder.

*Interface* - In Java, an interface is a reference type similar to a class that specifies a set of abstract methods that a class implementing the interface must provide their own implementation of.

*Abstract methods* - An abstract method in Java is a method that is declared without an implementation.

Any class that implements an interface must provide their own implementation for the methods specified by that interface. This is a way to achieve abstraction which if you remember from COP 3330, abstraction is one of the key principles of OOP.

You can have methods in a class that have an interface as a parameter. You can then pass any object of a class that implements that interface and then use that class' implementation. Lambdas allow us to pass in a specific implementation of an abstract method as a parameter instead of an object with its own implementation. This can only be done when the interface has a single abstract method (sometimes called a SAM interface). This makes the code much more concise.

Without lambdas you would need to have a class that implements the interface, then provide an implementation for the abstract method of the interface, then create an object of that class which then we could pass into whatever method takes that interface as a parameter. This is a lot of extra work when all we really want to do is have a specific 1-time implementation for the abstract method.

Lambdas can be written out as follows:

```
(parameter1, parameter2) -> { /* code block implementing some abstract method */ }
```

If there is only 1 parameter, the parentheses can be dropped.

```
parameter1 -> { /* code block implementing some abstract method */ }
```

If the code block is a 1-line return statement the curly braces, return, and semicolon can be dropped.

```
(parameter1, parameter2) -> parameter1.compareTo(parameter2)
```

One example of an interface with a single abstract method is the Comparable interface. Below is an example of passing a Comparable to the Collections.sort method. The first is not using lambdas and the 2nd which does. Notice how the "public int compare" part of the method gets dropped when implementing the function as a lambda.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        // Create a list of integers
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(5);
        list.add(20);
        list.add(15);

        // Create a comparator to sort the list in descending order
        Comparator<Integer> comparator = new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return o2.compareTo(o1);
            }
        };

        // Sort the list using the comparator
        Collections.sort(list, comparator);

        // Print the sorted list
        for (Integer i : list) {
            System.out.println(i);
        }
    }
}
```

With lambda

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        // Create a list of integers
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(5);
        list.add(20);
        list.add(15);

        // Sort the list using the comparator
        Collections.sort(list, (o1, o2) -> {return o2.compareTo(o1);});

        // Print the sorted list
        for (Integer i : list) {
            System.out.println(i);
        }
    }
}
```