CS 361: Functional Programing:

Functional programming (20 minutes)

Question to Class: We've been doing Python for a while now. What's one of the biggest uses of Python? Like we learned over the first few weeks - especially what we learned with Matt.

- Data analysis

So recently we're dealing Data analysis. Input > Some output.

And more recently we do this analysis on big Data**. What do we need to do for Big Data processing?** *Parallel Programming* which Java ain't great with. (*Ch 1.1 – 1.2*)

If languages don't adapt to changing reequipments – people no use, so it would have died. Java didn't want to die, Java changed. So – we now get to Java 8 and newer introduced these new concepts to be able to parallel processing of Big Data:

- Stream Processing
  - "Steam" – A sequence of data items that are produced one at a time;
  - The point is that, this concept helps with the implementation of parallelism for free, almost.
  - **If we're doing stuff in parallel what's a problem that could arise?**
    - Answer: Mutating data, we cannot do that.
- Behavior Parametrization
  - Ability to pass code to an API
  - Before it was only primitives or objects
- No data mutation & Behavior Parametrization are the cornerstone of functional programming

To implement these concepts that allows parallel processing of big data, Java introduced these new features (*Ch 1.1 – 1.2*):

- Streams API – It's a new API. Of many features - it supports parallel processing.
  - There's only one rule – we covered this in Operating System as well.
  - In summary: And this is another reason why we use Functional Programming - it enables parallel processing of data – BIG Data.
  - Makes use of the idea of behavior as a parameter
- In addition, in Streams another change in Java 8, was the introduction of lambdas. Today's topic. Helps with using Streams. Helps make use of Streams.
- **Ask Chris: What are you covering in your lesson** April 3rd**.?**
  - There are also other changes made to Java that help w/concurrency on top of the Streams API

Another benefit: Why do functional programming? (10 minutes)

There's another benefit that comes from this idea of functional programming. You hopefully will see this as we do the exercises but. It's the same reason as the one of the answers to this question:

Question to Class: Going on what you already know about Java: What features in Java make it a popular programing language? – We talked about this first day of class.

- If they don't get it, ask: Do you need to write your own sorts?
- A robust standard library. For example:
- Collections -> allow sorting
- You're transferring code to better programmers, programmers who wrote code that's more efficient. You don't need to write it. You don't need to care how they implement it. You just sort.

And for this same reason – we use functional programming. We can focus on the "what" and not the "how". Hopefully that's what the exercise will show. But before that.

Java's not the only language to do functional programming – if you recall in week 2, what language introduced functional programming? Lisp was the original language to do this. Other languages like JS have this. Scala too – which y'all will learn about April 10th. Team's been tasked to do this WITH Java.

Exercises:

Let's do some coding – a bit of data processing:

- Explain the BlueJ – mention how we parsing a .csv file…data processing…team python…data we're processing isn't big, nor going to be done with parrel – but with steams you can
  - Write the properties on the board
- Exercise 1: Write a program to filter out Apples that are Green. (*Ch 2.1*)
  - How - They should be able to write a for-each loop, since our class tolerates for each loop

```
public List<Apple> filterGreenApples() {
    List<Apple> result = new ArrayList<>();

    for (Apple apple : apples) {
        if (apple.getColor().equals("Green"))
            result.add(apple);
    }

    printList(result);
    return result;
}
```

  - **Ask question: What if you now want to filter out Apples that are Red? Or even Yellow?**
- Exercise 2: Then by color given in a parameter. (*Ch 2.1*)
  - How – They should copy and paste the code from Ex. 1 but instead of "Green" use the argument.

```
public List<Apple> filterByColor(String color) {
    List<Apple> result = new ArrayList<>();

    for (Apple apple : apples) {
        if (apple.getColor().equals(color))
            result.add(apple);
    }

    printList(result);
    return result;
}
```

- **Ask: What if we wanted to filter by the Apple's weight?**

- Exercise 3: Then by weight. (*Ch 2.1*)
  - How – Like Ex. 1 & Ex. 2; purposeful copy-paste error

```
public List<Apple> filterByWeight(int weight) {
    List<Apple> result = new ArrayList<>();

    for (Apple apple : apples) {
        if (apple.getWeight() > 110)
            result.add(apple);
    }

    printList(result);
    return result;
}
```

  - **Ask: What if we wanted to filter by the Apple's weight and color or something else?**
  - The point? Not DRY Code and we have to know how loops work and the person who makes the method needs to know requirements. Not abstract. Now let's start a little bit of concept of behavior parametrization. Copy-paste errors
- Example 4: Let's try behavior parametrization. (*Ch 2.2*)
  - Make the Interface – We call it Predicate, because predicates return booleans

```
public interface ApplePredicate {
    public boolean test(Apple a);
}
```

- Make the behavior implementing the Interface

```
public class AppleColorPredicate implements ApplePredicate {
        public boolean test(Apple apple){
            return "Green".equals(apple.getColor());
        }
}
```

- Make the filter() method that takes in behavior

```
public List<Apple> filter(ApplePredicate p) {
 List<Apple> result = new ArrayList<>();
  for (Apple apple : apples) {
   if (p.test(apple)) {
    result.add(apple);
     }
   }
   printList(result);
   return result;
}
```

- Still we're repeating code…less then before…we still gotta deal with names and shit and for what? Something that we're gonna use only once. Still verbose. This bring us to lambdas.

Lesson about lambdas! (*Ch 3.1 - 3.2I*):

Structure:

- (parameters) -> expression; Or
- (parameters) -> { statements;}

Expression is the return value or what you want the lambda to do.

Lambdas are:

- Anonymous – meaning that they have no name
- Function – it's not associated with a class
- Passed around – they can be used as arguments and be able to be stored
- Concise – less boilerplate needed (pg 44)

Example 5: Lambda with green. (*Ch 2.3/Ch 3.1 - 3.2*)

```
(Apple apple) -> apple.getColor().equals("Green")
```

- Note step by step; first we are taking care of the parameter
- Expression is what we want to – in this get return a boolean after doing a test

Example 6: Lambda with Yellow (*Ch 2.3/Ch 3.1 - 3.2*)

```
(Apple apple) -> apple.getColor().equals("Yellow")
```

Exercise 7: Then use the Predicate with lambdas. (*Ch 2.3/Ch 3.1 - 3.2*)

- How to do for weight > 100 (Apple a1) -> a.getWeight() > 100;
  - Show we could do Ex 1 - 3, but with different calls.

At this point, we covered the basics, now the more complex – i.e. less work we have to do:

- Functional Interfaces in Java (Ch 3.4) – We don't have to write our own predicate – import the predicate and delete the one we have: java.util.function.Predicate &

```
public List<Apple> filter(Predicate<Apple> p) {
    List<Apple> result = new ArrayList<>();

    for (Apple apple : apples) {
        if (p.test(apple))
            result.add(apple);
    }

    printList(result);
    return result;
}
```

  - We can do more!
- Type discussion (3.5) – We don't have to put Apple in "(Apple a1)", sample runs
- MAYBE: Method references (3.6) - this is still unclear to me; I might not get to it either. L.

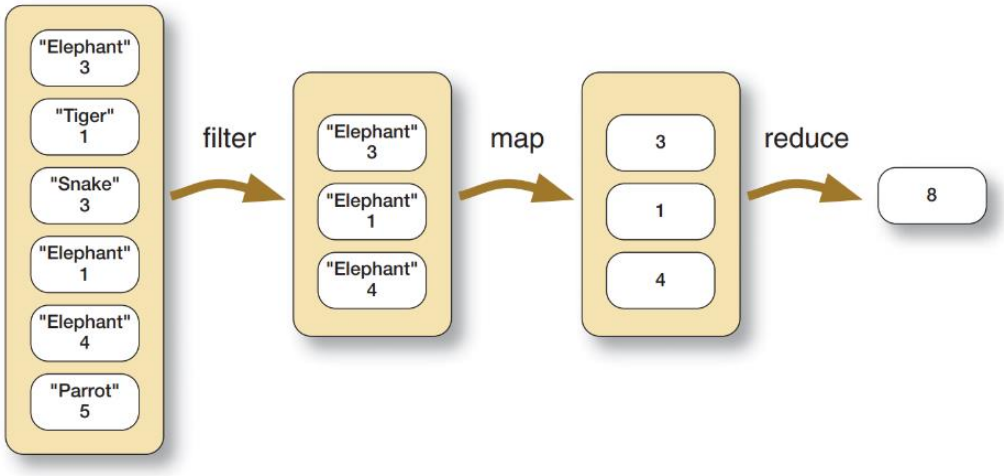
One more thing: we didn't need to write our own filter even: There's one built in with the Streams's API:

```
apples.stream().filter((apple) -> apple.getWeight() > 100);
```

Here we can see a summary.

- We see behavior as a parameter – one of the cornerstones of functional programming – in the form of lambdas
- We can change what we want to do by varying the lambda expression that's send into a method. Adapts to changing requirements.
  - Changing the lambdas are less error prone from copy paste errors.
  - We are literally sending the "what" we want to do as a parameter. It allows us focus on the what. We don't need to care about how the filter() is implemented.
  - BTW even for the people who create methods like filter() are benefited. They by no means could know all the possible requirements people want to filter() by, so instead making the people using the class send the behavior saves work.
- And looking at the lambdas show what we want to do. Just read "We want to filter what? Apples that weight more then 100". Concise.
- PLUS what do we achieve by filtering? We are starting to do data processing – the main use of functional programming…there's much much more.

Figure 5.4
A pipeline of stream functions



But for that, I pass it along to Simone. Make sure to read chapter 3 of the textbook in case you didn't learn anything today - pay attention to Simone's lecture cause Tyler will be combing lambdas and Stream together the week after on the 27th.

Homework

In class we followed through an example following a list of Apples. Now you will filter a list of Cars or a list of Students.

- Team Python: Cars (Download FunctionalProgCar.zip)

- Team Prolog: Students (Download FunctionalProgStudent.zip)

- Team Java: Apples

The respective .zip files already have what we covered in class. Only need to write lambdas.

In a **PDF** upload the answer as well as the lambdas you used to find what the questions are asking for.

Example: Team Java:

Apples		
Property	Type	Range
weight	double	60.0 – 120.0
color	String	"Red", "Green", "Yellow"
numberSeeds	int	3 – 7 (inclusive)
isPoisonous	boolean	True, False

Using lambdas and the filter() method, find the following information. Please include the lambda expression you used.

1. The number of Apples that are Red.
    a. Hint: write a lambda that would filter out Red Apples.
2. Find the Apple that is Poisonous. What are its properties?
3. The number of Apples that are Yellow and weigh more than 100.00.
4. Convert the printList() method in your program to use lambdas.
    a. Hint: Use the forEach() method

Sample Answer (as an example):

1. Answer: 492
   Lambda: (apple) -> apple.getColor().equals("Red")
2. Answer: 1, Apple{color='Yellow', weight=84.03, seeds=4, isPoisonous=true}
   Lambda: (apple) -> apple.getIsPoisonous()
3. Answer: 89
   Lambda: (apple) -> apple.getColor().equals("Yellow") && apple.getWeight() > 100
4. See code:

```
private void printList(List<Apple> appleList) {

    System.out.println("List of Apples: ");

    appleList.forEach( apple -> System.out.println(" - " + apple));

    System.out.println("Filtered list length: " + appleList.size());

}
```

Team Python (Download "FunctionalProgCar..zip"):

Car		
Property	Type	Range
weight	double	2600.0 – 4400.0
brand	String	"Audi", "Volvo", "Nissan", "Toyota", "BMW"
rating	int	5 – 10 (inclusive)
isElectric	boolean	True, False

Using lambdas and the filter() method, find the following information. Please include the lambda expression you used.

1. The number of Cars that are Volvo.
    a. Hint: write a lambda that would filter out Cars that are Volvo.
2. The number of Cars that are electric. Pick of the any and give its properties.
3. The number of Cars that are Audi and weigh more than 3500.0.
4. Convert the printList() method in your program to use lambdas.

Team Prolog (Download "FunctionalProgStudent.zip"):

Student		
Property	Type	Range
gpa	double	1.0 – 4.0
major	String	"Computer Science", "Education", "Math", "English"
numClasses	int	3 – 6 (inclusive)
isMasters	boolean	True, False

Using lambdas and the filter() method, find the following information. Please include the lambda expression you used.

1. The number of Students that are Education majors.
    a. Hint: write a lambda that would filter out Students that are Education majors.
2. The number of Students that are doing their Masters. Pick of the any and give its properties.
3. The number of Students that are Math and have a GPA more than 3.50.
4. Convert the printList() method in your program to use lambdas.

Answer key:

Car:

1) Answer: 191
   (car) -> car.getBrand().equals("Volvo")
2) Answer: 327 (anything IsElectric is True)
   (car) -> car.getIsElectric()
3) Answer: 94
   (car) -> car.getBrand().equals("Audi") && car.getWeight() > 3500
4) Answer:

```
private void printList(List<Car> carList) {

    System.out.println("List of Car: ");

    carList.forEach( car -> System.out.println(" - " + car));

    System.out.println("Filtered list length: " + carList.size());

}
```

Student:

1) Answer: 189
   (student) -> student.getMajor().equals("Education")
2) Answer: 517; (anything isMasters is true)
   (student) -> student.getIsMasters()
3) Answer: 35;
   (student) -> student.getMajor().equals("Math") && student.getGPA() > 3.5
4) Answer:

```
private void printList(List studentList) {

        System.out.println("List of Students: ");

        studentList.forEach( student -> System.out.println(" - " + student));

        System.out.println("Filtered list length: " + studentList.size());

}
```