Pro Dev Session

Pro Dev Session

You may want to write this down.

Collaboration is an essential part of being a great teammate and developer. To be a good collaborator, we must master our collaboration tools, namely git and GitHub.

Today we will get more practice with git. We'll be building on this for the next several lessons.

Pro Dev Session

You may want to write this down.

Today we’ll gain a better understanding of branching and code review. We’ll be learning two new command.

The first command is `git push <remotename> <branchname>`.

The second command is `git fetch <remotename> <branchname>`.

Fetch allows us to get a copy of a branch from GitHub without merging it into our code.

Pro Dev Session

You may want to write this down.

Below is a typical git workflow:

- You are assigned a task like create a login or fix a bug.
- You create a new branch in your local repository with a descriptive name `git branch login`.
- You checkout this new branch `git checkout login`.
- You work. You save as you go as appropropriate `git add -A` , `git commit -m "login validation complete"`.
- You merge in the latest version of the remote master to check for merge conflicts `git pull origin master`.
- You complete your feature and add and commit one last time. You push these changes `git push origin login`.
- On GitHub, you create a pull request.

Pro Dev Session

You may want to write this down.

After you create a pull request a teammate will typically review your code. Today we will focus heavily on this review portion.

You teammate should begin by fetching a copy of your branch. You are familiar with the git command `pull`. We often use `git pull origin master`, to merge the latest version of the remote master into our repository.

The `pull` command is actually a combination of fetch (which brings a copy of the remote's branch to our local machine) and `merge` which merges this code with our current working branch.

Pro Dev Session

You may want to write this down.

When we are performing code review, we want to get a local copy of a branch but NOT merge it into our current working code. For this, we use `git fetch origin <branchname>`.

This will allow us to run the code and manually test it during our code review.

Git Practice

Work in PAIRS to complete all of the goals below.

Goals:

- Create a new repository. Each teammate should clone it.
- Create a new IntelliJ project. Add, commit, and push to master.
- Each teammate should update their local master.
- Each teammate should create a new branch and add a class - one representing a student and one representing a class.
- Add, commit, push, and create a pull request.
- Use git fetch origin <branchname> and to pull your partner's code to your machine.
- Use git checkout <branchname> to switch to the new branch.
- Manually test the code thoroughly. Can you break it? Does it need any additional error handling? Should you overload any methods? If so, leave a comment in the pull request. If not, merge the pull request.

20 minutes!

Stand Up!

Lambdas & Streams

Lambdas& Streams

EFFICIENT AGGREGATION

Notebooks Ready? It’s time for a brief (but not that brief) lecture.

Lambdas& Streams

EFFICIENT AGGREGATION

Warning: This section is going to get tricky.

You are about to see a lot of new syntax, new concepts, and new terminology.

If this feels overwhelming, that’s perfectly normal and okay. You don’t need to master this today. Or even this week. This is something you can practice for the remainder of the course without falling behind. There will not be tests on this particular subject.

It is, however, an important part of the current Java landscape, so you should make an effort to learn it.

Lambdas& Streams

EFFICIENT AGGREGATION

Bare with me for the next couple of slides. This may get rocky.

Lambdas& Streams

EFFICIENT AGGREGATION

Let’s start with the what:

What are streams?

Stream is an interface. It has some really cool methods.

You can use these methods on Collections like ArrayLists… but how?? ArrayList doesn’t implement the Stream interface??

You can call the .stream() method on an ArrayList to create a kind of stream wrapper around the ArrayList that allows you to use these new methods.

Lambdas& Streams

EFFICIENT AGGREGATION

Let’s start with the what:

What are lambdas?

Lambda expressions are basically a new more concise syntax for methods in certain context that allow methods to be passed as parameters.

Check-in Time

What does it mean to pass a method as a parameter?

Lambdas& Streams

EFFICIENT AGGREGATION

Now let’s look at the why:

Why are we learning about lambdas and streams?

Firstly, we are learning about them together because they play very nicely together. You’ll almost always see lambda expressions used with streams.

Secondly, we are learning about lambdas and streams because they make performing certain operations on Collections significantly easier.

Lambdas& Streams

EFFICIENT AGGREGATION

Let’s look at an example.

Suppose we have a list of Motorcycles. Each Motorcycle has maxSpeed, weight, cc’s, color, make, model, and vin.

Now we need a new list of only the fast Motorcycles, say only those that reach a top speed of 180 mph or greater.

Lambdas & Streams

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
public List<Motorcycle> getFastBikes(List<Motorcycle> motoList){

        List<Motorcycle> fastList = new ArrayList<>();

        for (Motorcycle moto: motoList){
                if(moto.getMaxSpeed() >= 180){
                        fastList.add(moto);
                }
        }

        return fastList;

}
```

Now let’s refactor with lambdas and streams.

Lambdas & Streams

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
public List<Motorcycle> getFastBikes(List<Motorcycle> motoList){

        return motoList
            .stream()
            .filter(moto -> moto.getMaxSpeed() <= 180)
            .collect(Collectors.toList());

}
```

Lambdas & Streams

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
public Map<String,Motorcycle> getByMake(List<Motorcycle> motoList){

         Map <String,Motorcycle> makeMap = new HashMap<>();

         for(Motorcycle moto: motoList){
                  if(makeMap.get(moto.getMake())== null){
                           makeMap.put(moto.getMake(), new
ArrayList<>(Arrays.asList(moto)));
                  } else {
             makeMap.get(moto.getMake()).add(moto);
         }
         }

         return makeMap;
}
```

Lambdas & Streams

WATCH & LEARN

Close your laptop. Eyes on my screen. Pay attention.

```
public Map<String,Motorcycle> getByMake(List<Motorcycle> motoList){

          return motoList
               .stream()
                    .collect(Collectors.groupingBy(moto -> moto.getMake()));


}
```

Now let’s dissect these crazy new tools

Lambdas& Streams

EFFICIENT AGGREGATION

We start by calling the stream method on our Collection to create a Stream and access the Stream methods.

```
motoList
     .stream()
     .filter(moto -> moto.getMaxSpeed() <= 180)
     .collect(Collectors.toList());
```

Lambdas& Streams

EFFICIENT AGGREGATION

Next we can call zero or any number of intermediate operations. Intermediate operations are methods that can be called on a Stream that also return a Stream.

```
motoList
      .stream()
      .filter(moto -> moto.getMaxSpeed() <= 180)
      .collect(Collectors.toList());
```

Because they return a Stream, you can then call another intermediate operation. They can be chained together in this way.

Lambdas& Streams

EFFICIENT AGGREGATION

Many intermediate operations take a method as a parameter. For this we use lambda expressions.

```
motoList
     .stream()
     .filter(moto -> moto.getMaxSpeed() <= 180)
     .collect(Collectors.toList());
```

This deserves its own slide.

Lambdas& Streams

Suppose we have a method:

```
boolean filterByMake(Motorcycle moto) {
  return moto.getMake().equals("Suzuki");
}
```

With lambda syntax we can remove the name and return type (this is inferred):

```
(Motorcycle moto) -> {
  return moto.getMake().equals("Suzuki");
}
```

If there is only one parameter, we can remove parentheses and type (this is inferred):

```
moto -> {
  return moto.getMake().equals("Suzuki");
}
```

And in the event that the method body is a single line, we can remove the curly braces and return keyword:

```
moto -> moto.getMake().equals("Suzuki");
```

Lambdas& Streams

EFFICIENT AGGREGATION

The highlighted portion should now make a little more sense to you:

```
motoList
     .stream()
     .filter(moto -> moto.getMaxSpeed() <= 180)
     .collect(Collectors.toList());
```

Lambdas& Streams

EFFICIENT AGGREGATION

The highlighted portion should now make a little more sense to you:

```
motoList
     .stream()
     .filter(moto -> moto.getMaxSpeed() <= 180)
     .collect(Collectors.toList());
```

Lambdas& Streams

EFFICIENT AGGREGATION

Every pipeline ends in a terminal operation:

```
motoList
     .stream()
     .filter(moto -> moto.getMaxSpeed() <= 180)
     .collect(Collectors.toList());
```

Terminal operations result in a non-stream value like an Integer, a List, a Map, or a Boolean. Additional Stream operations cannot be chained to the pipeline after a terminal operation. It terminates the pipeline.

Lambdas& Streams

EFFICIENT AGGREGATION

There are a fair number of intermediate and terminal operations, but here's a list to get you started:

- `filter` - An intermediate operation that filters the object in a stream based on the logic in the given lambda expression.
- `mapToXxx` - A family of intermediate operations that map an input stream of one type into a stream of another type based on the logic in the given lambda expression. For example, `mapToInt` would map an incoming stream of object to a stream of integers.
- `forEach` - A terminal operations that acts on each object in a stream according to the logic in the given lambda expression.
- `collect` - A terminal operation that returns a collection of objects according to its parameter.
- `Average` - A terminal operation that returns the average of the given input stream. The average is returned as an `OptionalDouble`.
- `getAsXxx` - This is a family of operations that take an `OptionalXxx` type as input and then return the value as an `Xxx`. For example (from above), the average aggregate operation returns an `OptionalDouble`, so we would use `getAsDouble` to convert the `OptionalDouble` to `Double`.

Lambdas& Streams

CODE-A-LONG

Open your laptop. Code with me. Don't jump ahead.

Follow along with me as I complete this activity.

Goals:

- Filter an ArrayList of Motorcycles so we only get bikes that weigh less than 500 pounds.

Lambdas& Streams

CODE-A-LONG

Open your laptop. Code with me. Don't jump ahead.

Follow along with me as I complete this activity.

Goals:

- Create a Map that groups Motorcycles by cc's.

Stay Seated & Take 3 Deep Breaths.

RELAX.

Now take a short walk. Clear your head. After a few minutes break, quickly review your notes. We'll start back shortly.

Lambdas & Streams

Lambdas& Streams

CODE-A-LONG

Open your laptop. Code with me. Don’t jump ahead.

Follow along with me as I complete this activity.

Goals:

- Find the average topSpeed.

Lambdas& Streams

CODE-A-LONG

Open your laptop. Code with me. Don’t jump ahead.

Follow along with me as I complete this activity.

Goals:

- Find the maximum topSpeed.

Lambdas& Streams

INDEPENDENT PRACTICE

It’s time to fly. Focus. Work hard. Ask for help when you need it.

Work in PAIRS to complete all of the goals below.

Goals:

- Given an ArrayList of Strings, create a new ArrayList of only strings starting with D (case sensitive) or B (case sensitive) .

15 minutes!

lunch.

Project Time