Streams and lambdas

Lambdas

- An abstraction for an equation, some examples of using lambdas:
  `(int x, int y) -> {return x+y;}` x -> x*x
  `() -> x (supplier)
- Lets say we have a student object with parameters (Name, Major, Campus)
- If we want to print all the students in a lists info we can write the following
  `students.forEach(` x-> print(x.name() + ' ' +x.major() + ' ' + x.campus()));
- If we want to filter for only Vancouver campus students:
  `students.removeIf(student -> student.campus != "Vancouver");
- Sort them by name:
  `students.sort((a, b) -> a.name.compareTo(b.name)));
- We can specify parameters if we want
- Use custom comparators for lambdas
  `Comparator<> lengthCompare = (s1, s2) -> {return s1.length() - s2.length();}
- functional interface: an interface with **exactly one abstract method**, helpful with lambda expressions
  - Java creates these interfaces when we are calling lambda expressions
  - An example of using a functional interface

    `public static int applyOperation(int number, Operation operation)`
    `{` return operation.perform(number);
    `}` @FunctionalInterface
    `interface Operation {` int perform(int number);
    `}` main() {
    `int result = applyOperation(5, n-> n * 2);` }
  - Here we pass to things to applyOperation, 5 and a lambda expression
  - Since Operation is a func interface then we can use a lambda expression in its place
- We **don't** give a name to the lambda expression
- We don't make functional interfaces, Java does that for us

functional programming

- Cannot mutate inputs
- Allows us to write easier to understand code

- Allows us to focus on problem rather than code
- Helps with parallelism

Streams

- Allows us to write a lot less code
- We can use lambda expressions in the code alongside with streams to make our lives much easier
- We can do a lot with streams such as
  - Reduce: reduce the elements in a collection
  - Map: map the elements in a collection to another collection
  - Filter: filter a collection
- Going back to students we can do the following, `students.stream().map(x -> x.name())` .forEach(x -> print(x));
- We can map create a map of students to their major, very easily
- We can collect all the stream elements to a list using .collect(Collectors.toList)
- Lets say we have a list of books ad want to get a list of all the names of the authors over the age of 50 in all uppercase `ls.stream().map(book -> book.getAuthor()).filter(author -> author.getAge() >= 50)` .map `getLastname` `.map toUpperCase` .collect(toList());
- :: is a method reference