In-Class Exercise: Lambda Expressions

- Work in your group
  - The goal is to get *everyone* in your group on board
- Fully worked example is from the Oracle Java Tutorials:
  - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- See how much code you can develop
- Focus on the relationship between each step
  - We'll do steps 1-7 and 9

Preliminaries

```
public class Person {
   public enum Sex { MALE, FEMALE }
   String name;
   LocalDate birthday;
   Sex gender;
   String emailAddress;
   public int getAge() { // ... }
   public void printPerson() { // ... }
}
```

- Let’s implement actions on select Person objects in a static context where we have a roster:

```
List<Person> roster;
```

Approach 1: Create Methods That Search for Members That Match One Characteristic

Create a static method that prints members older than a certain age:

```
public static void printPersonsOlderThan(
   List<Person> roster, int age)
```

Approach 1: Solution

Create a static method that prints members older than a certain age:

```
public static void printPersonsOlderThan(
   List<Person> roster, int age)  {
     for (Person p : roster) {
       if (p.getAge() >= age) {
          p.printPerson();
       }
     }
  }
```

Approach 2: Create More Generalized Search Methods

Now print members within a specified range of ages

```
public static void printPersonsWithinAgeRange(
   List<Person> roster, int low, int high)
```

Approach 2: Solution

Now print members within a specified range of ages

```
public static void printPersonsWithinAgeRange(
   List<Person> roster, int low, int high) {
   for (Person p : roster) {
     if (low <= p.getAge() && p.getAge() < high){
        p.printPerson();
     }
   }
}
```

Approach 3: Specify Search Criteria Code in a Local Class

Now print members that satisfy a general test

```
public static void printPersons(
   List<Person> roster, CheckPerson tester)  {
   for (Person p : roster) {
      if (tester.test(p)){
         p.printPerson();
      }
    }
 }
```

Define `CheckPerson`, implement with named class that filters members eligible for Selective Service (males, 18 to 25), and call this method

Approach 3: Solution

```
interface CheckPerson {
   boolean test(Person p);
}
class CheckPersonEligibleForSelectiveService
   implements CheckPerson {
   public boolean test(Person p) {
      return p.gender == Person.Sex.MALE
           && p.getAge() >= 18
           && p.getAge() <= 25;
   }
}
printPersons(roster,
   new CheckPersonEligibleForSelectiveService());
```

Approach 4: Specify Search Criteria in an Anonymous Class

Replace named class with an anonymous class:

```
printPersons(
   roster,
   ?????








 );
```

Approach 4: Solution

Replace named class with an anonymous class:

```
printPersons(
   roster,
   new CheckPerson()  {
      public boolean test(Person p) {
         return p.gender == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25;
      }
    }
 );
```

Approach 5: Specify Search Criteria with a Lambda Expression

Replace anonymous class with lambda expression:

```
printPersons(
    roster,
    new CheckPerson()  {
       public boolean test(Person p) {
          return p.gender == Person.Sex.MALE
             && p.getAge() >= 18
             && p.getAge() <= 25;
       }
     }




 );
```

Approach 5: Solution

Replace anonymous class with lambda expression:

```
printPersons(
   roster,
   (Person p) ->
            p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25
 );
```

Approach 6: Use Standard Functional Interfaces with Lambda Expressions

Reconsider the `CheckPerson` interface:

```
interface CheckPerson {
     boolean test(Person p);
}
```

`java.util.Function` **defines:**

```
interface Predicate<T> {
     boolean test(T t);
}
```

Rewrite printPersons and make the call:

```
public static void printPersonsWithPredicate(…) {…}
```

Approach 6: Solution

```
public static void printPersonsWithPredicate(
    List<Person> roster,
    Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p) {
            p.printPerson();
        }
    }
 }

 printPersonsWithPredicate(roster,
    p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <=25
 );
```

Approach 7: Use Lambdas Throughout Your Application

What other function could we pass around?

```
    printPerson(p);
```

What is the appropriate functional interface?

```
interface Consumer<T> {
     void accept(T t);
}
```

Rewrite printPersons and make the call:

```
public static void processPersons(…) {…}
```

Approach 7: Solution

```
public static void processPersons(
   List<Person> roster,
   Predicate<Person> tester,
   Consumer<Person> block) {
   for (Person p : roster) {
       if (tester.test(p) {
          block.accept(p);
   }  }  }

 processPersons(roster,
    p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <=25,
    p -> p.printPerson()
 );
```

More Approach 7: Add a filter

Instead of printing the whole Person, how about just the email?

Rewrite processPersons and make the call:

```
public static void processPersonsWithFunction(
   List<Person> roster,
   Predicate<Person> tester,
   Function<Person, String> mapper,
   Consumer<String> block)
{…}
```

More Approach 7: Solution

```
public static void processPersonsWithFunction(,,,) {
   for (Person p : roster) {
       if (tester.test(p) {
          String data = mapper.apply(p);
          block.accept(data);
   }  }  }

 processPersonsWithFunction(
         roster,
    p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <=25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
 );
```

Approach 9: Use Aggregate Operations

```
roster.stream()
    .filter(
          p -> p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```