Problem Set 4 Solution

Lambda Expressions

1. For each of the following expressions, tell whether it is valid or not. If valid, explain the reasoning. If not valid, explain why.

   1. `() -> { }`

      ANSWER

      Valid. Corresponds to a method that takes no arguments, returns void, and has an empty body, e.g.

      `public void stuff() { }`

   2. `() -> "Hello"`

      ANSWER

      Valid. Corresponds to a method that takes no arguments and returns a `String`:

      `public String stuff() { return "Hello"; }`

   3. `() -> { return "Goodbye"; }`

      ANSWER

      Valid. Similar to previous, except it's written as an explicit statement inside curly braces.

      `public String stuff() { return "Goodbye"; }`

   4. `(Integer i) -> return i+10;`

      ANSWER

      Invalid. Since `return` is a control flow statement, it has to be enclosed within braces.

   5. `(String s) -> { "Bourne Ultimatum"; }`

      ANSWER

      Invalid. `"Bourne Ultimatum"` is an expression, not a statement. You can do either of the following to get a correct lambda expression:

      - Move expression out of the braces:

        `(String s) -> "Bourne Ultimatum"`

      - Do a return statement:

        `(String s) -> { return "Bourne Ultimatum"; }`

2. Which of the following are functional interfaces?

   1.
      ```
      public interface Sum1 {
         int sum(int i, int j);
      }
      ```

      ANSWER

      Yes.

   2.
      ```
      public interface Sum2 extends Sum1 {
         double sum(double i, double j);
      }
      ```

      ANSWER

      No. `Sum2` has two methods.

   3.
      ```
      public interface Rectangle {
         double getWidth();
      ```

```
   double getHeight();
   default double area() {
      return getWidth()*getHeight();
   }
}
```

ANSWER

No. There are two abstract methods.

---

3. Which of the following are valid uses of lambdas?

   1.
      ```
      public interface Executor {
          void execute();
      }
      public void do(Executor ex) {
          ex.execute();
      }
      do(() -> { });
      ```

      ANSWER

      Yes. The lambda takes no args and returns nothing, which matches the `execute` method of the `Executor` interface.

   2.
      ```
      public interface Proc<T> {
          T process();
      }
      public Proc<String> get() {
          return () -> "I am a go getter!";
      }
      ```

      ANSWER

      Valid. The lambda in the return takes no args and a `String`, which matches the `process` method of the `Proc` interface, with the binding of `String` to the generic type `T`.

   3. `Predicate<Student> p = (Student s) -> s.getMajor();`

      ANSWER

      Invalid. The lambda should return a boolean.

   4. `BiFunction<Integer,Integer,String> bif = (int i, int j) -> ""+i+j;`

      ANSWER

      Invalid. The args for the lambda must be `Integer`s. Auto conversion to `int` will not be done. (If you omit the data type for the arguments, it will work just fine.)

---

4. This question refers to the `Student` class presented in lecture (see Sakai -> Resources -> Feb 16 -> `Student.java`)

   1. Write a NAMED lambda expression using a method reference to check if a student is a senior.

      ANSWER

      `Predicate<Student> is_senior = Student::isSenior;`

   2. Write a NAMED lambda expression using a method reference to get the major of a student.

      ANSWER

      `Function<Student,String> major = Student::getMajor;`

   3. Given the following filter method:

      ```
      public static List<T>
      filter(List<T> list, Predicate<T> p) {
         List<T> res = new ArrayList<T>();
         for (T t: list) {
            if (p.test(t)) {
               res.add(t);
            }
      ```

```
    }
    return res;
  }
```

For each of the following, write one or more `Predicate` instances as NAMED lambda expressions that can be passed to the `filter` method to get the required set of students. (Note: when composing predicates, you want to use named lambda expressions in the composition, otherwise the syntax gets unwieldy/unacceptable.)

1. All non-CS majors

   ANSWER

   ```
   Predicate<Student> cs_major = s -> s.getMajor().equals("CS");
   Predicate<Student> non_cs_major = cs_major.negate();
   ```

2. All CS and Physics majors who are commuters

   ANSWER

   ```
   Predicate<Student> physics_major = s -> s.getMajor().equals("Physics");
   Predicate<Student> commuter = Student::getCommuter;
   Predicate<Student> pred = (cs_major.or(physics_major)).and(commuter);
   Predicate<Student> is_senior = Student::isSenior;
   ```

3. Math seniors who are not commuters

   ANSWER

   ```
   Predicate<Student> math_major = s -> s.getMajor().equals("Math");
   Predicate<Student> pred = (math_major.and(is_senior)).and(commuter.negate());
   ```

4. Resident non-Math non-freshman students

   ANSWER

   ```
   Predicate<Student> is_freshman = Student::isFreshman;
   Predicate<Student> pred = commuter.negate().and(math_major.negate()).and(is_freshman.negate());
   ```