NUS | Computing
National University of Singapore

NUS WebMail IVLE LIBRARY MAPS

Search search for... in NUS Websites GO

CodeCrunch

Home | My Courses | Browse Tutorials | Browse Tasks | Search | My Submissions | Logout | Logged in as: **e0959123**

CS2030 (2310) Practical Assessment #2

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

CS2030 Practical Assessment #2

One of the earliest design principles taught to CS2030 students is "Tell-Don't-Ask", in which data encapsulated in objects are hidden from the client while only exposing appropriate services that operate on them. However, with the inclusion of the `toString` method, a handful of students *unintentionally* work around "Tell-Don't-Ask" by extracting data from the `String` that is returned by the method. As an example, given the following `Point` class:

```
class Point {
    private final double x;
    private final double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}

jshell> System.out.println(new Point(1.0, 1.0).toString())
(1.0, 1.0)
```

One can retrieve the x-coordinate using

```
jshell> Double.parseDouble(
   ...> new Point(1.0, 1.0).toString()
   ...> .replaceAll("[() ]","")
   ...> .split(",")[0])
```

please don't do this...

After much deliberation, we have decided to restrict students from defining an overriding `toString` method, or any methods that returns `String`. From now on, students will define the `Point` class as follows:

```
class Point implements Stringable {
    ...
    public Str toStr() {
        return Str.of("(" + this.x + ", " + this.y + ")");
    }
}
```

A class should implement the `Stringable` interface that specifies the method `toStr()` that returns a `Str` object.

```
interface Stringable {
    public Str toStr();
}
```

The `Str` class is a special class that encapsulates a `String` but does not allow for it to be returned; in other words, there are no non-private methods that return a `String`. To print an object, the following will be used instead:

```
jshell> new Point(1.0, 1.0).toStr().run(x -> System.out.println(x))
(1.0, 1.0)
```

Task

In this task, you are to design the `Str` class, primarily to address the issue of adherence to "Tell-Don't-Ask". You are given the `Stringable` interface, as well as the `Point` and `Circle` classes. DO NOT modify these programs.

Take Note!

This task comprises a number of levels. You are required to complete ALL levels.

The following are the constraints imposed on this task. In general, you should keep to the constructs and programming discipline instilled throughout the module.

- Write each class or interface in its own file. Do not use single letter names for classes or interfaces.
- Ensure that ALL object properties and class constants are declared `private final`, unless otherwise specified.
- Ensure that your classes are NOT cyclic dependent.
- ONLY the following java libraries ARE allowed:
    - `java.util.Optional`
    - functional interfaces from `java.util.function`
- The following are NOT allowed:
    - `null`
    - `instanceof`
    - `if..else`, `switch..case`, `?:` conditional expression
    - `for`, `while`
    - `enum`
    - `Optional` methods: `isPresent`, `isEmpty`, `get` and its variants (`orElse`, `orElseGet`, `orElseThrow`), as well as `equals`
    - methods of the `String` class
- There is no need to use bounded wildcards.
- You are NOT allowed to define anonymous inner classes; define lambdas instead.
- Other usual restrictions:
    - Use only &&, `||` and `!` in logical expressions.
    - You are NOT allowed to use `*` wildcard imports.
    - You are NOT allowed to use method references `::`
    - You are NOT allowed to define array constructs, e.g. `String[]` or using ellipsis, e.g.`String...`
    - You are NOT allowed to use Java reflection, i.e. `Object::getClasses` and other methods from `java.lang.Class`

Level 1

Start by creating a `Str` class to support output. The `Point` class below illustrates how methods of the `Str` class are used.

```
class Point implements Stringable {
    private final double x;
    private final double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Str toStr() {
        return Str.of("(" + this.x + ", " + this.y + ")");
    }
}

jshell> new Point(1.0, 1.0)
$.. ==> Point@...
```

```
jshell> new Point(1.0, 1.0).toStr().run(x -> System.out.println(x))
(1.0, 1.0)
```

Write the `Str` class and include methods:

- `of` that takes in a `String` and encapsulates it in a newly created `Str` object;
- `run` that takes in an appropriate `Consumer` as an action and performs the given action on the `String` encapsulated in `Str`;
- any `private` constructors deemed necessary.

For ease of use, define another `print()` method that takes the place of the `run` method in the last test case above.

```
jshell> new Point(1.0, 1.0).toStr().print()
(1.0, 1.0)
```

Level 2

You are now given the `Circle` class that makes use of the `Point` class.

```
class Circle implements Stringable {
    private final Point centre;
    private final double radius;

    Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    public Str toStr() {
        return Str.of("Circle at centre ")
            .join(this.centre.toStr())
            .join(" with radius " + this.radius);
    }
}
```

Firstly, include the `map` and `flatMap` methods in the `Str` class. Both `map` and `flatMap` should only transform the encapsulated `String` into another `String`. Transforming to a value with a type other than `String` should not be allowed.

```
jshell> new Point(1.0, 1.0).toStr().map(x -> "$0 ==> " + x).print()
$0 ==> (1.0, 1.0)

jshell> new Point(1.0, 1.0).toStr().
   ...> flatMap(x -> Str.of("$0 ==> ").
   ...> map(y -> y + x)).
   ...> run(x -> System.out.println(x))
$0 ==> (1.0, 1.0)

jshell> new Point(1.0, 1.0).toStr().map(x -> x.length())
|  Error:
|  incompatible types: bad return type in lambda expression
|      int cannot be converted to java.lang.String
|  new Point(1.0, 1.0).toStr().map(x -> x.length())
|
```

Since we do not expect the client to know how to use `map` and `flatMap`, we need to facilitate joining to a `String` or `Str` object using `join` instead.

```
jshell> new Point(1.0, 1.0).toStr().join("...").print()
(1.0, 1.0)...

jshell> new Point(1.0, 1.0).toStr().join(Str.of("...")).run(x -> System.out.println(x))
(1.0, 1.0)...

jshell> new Point(1.0, 1.0).toStr().join(new Point(2.0, 2.0).toStr()).print()
(1.0, 1.0)(2.0, 2.0)

jshell> new Circle(new Point(0.0, 0.0), 1.0)
$.. ==> Circle@...

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().run(x -> System.out.println(x))
Circle at centre (0.0, 0.0) with radius 1.0
```

Level 3

We now make our `Str` class perform lazy evaluation by including an overloaded `of` method that takes in an appropriate `Supplier`.

```
jshell> Str.of("$0 ==> ").join(new Point(1.0, 1.0).toStr())
$.. ==> Str@...

jshell> Str.of("$0 ==> ").join(new Point(1.0, 1.0).toStr()).print()
$0 ==> (1.0, 1.0)

jshell> Str.of(() -> "$0 ==> ").join(new Point(1.0, 1.0).toStr())
$.. ==> Str@...

jshell> Str.of(() -> "$0 ==> ").join(new Point(1.0, 1.0).toStr()).run(x -> System.out.println(x))
$0 ==> (1.0, 1.0)

jshell> Str.of(() -> { System.out.print("beep..."); return "$0 ==> ";}).
   ...> join(new Point(1.0, 1.0).toStr())
$.. ==> Str@...

jshell> Str.of(() -> { System.out.print("beep..."); return "$0 ==> ";}).
   ...> join(new Point(1.0, 1.0).toStr()).
   ...> run(x -> System.out.println(x))
beep...$0 ==> (1.0, 1.0)

jshell> Str.of(() -> { System.out.print("beep..."); return "$0 ==> ";}).
   ...> join(new Point(1.0, 1.0).toStr()).
   ...> print()
beep...$0 ==> (1.0, 1.0)
```

Note that value caching is not required.

Level 4

To aid testing and development of the `Str` class, we shall include a debugging feature in `Str` that allows us to trace the formation of the `String`. As an example,

```
jshell> new Point(1.0, 1.0).toStr()
$.. ==> Str@...

jshell> new Point(1.0, 1.0).toStr().print()
(1.0, 1.0)

jshell> new Point(1.0, 1.0).toStr().trace()
traced Str: (1.0, 1.0)
(1.0, 1.0)

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr()
$.. ==> Str@...

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().print()
Circle at centre (0.0, 0.0) with radius 1.0

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().trace()
traced Str: Circle at centre // line 1
traced Str: (0.0, 0.0) // line 2
traced map: Circle at centre (0.0, 0.0) // line 3
traced flatMap: Circle at centre (0.0, 0.0) // line 4
traced map: Circle at centre (0.0, 0.0) with radius 1.0 // line 5
Circle at centre (0.0, 0.0) with radius 1.0 // same output as print()
```

Below is an explanation of the output of the last test above (with comments included separately) which is a trace of the `return` statement in the `toStr` method of the `Circle` class:

```
        return Str.of("Circle at centre ")
            .join(this.centre.toStr())
            .join(" with radius " + this.radius);
```

-
  ```
  traced Str: Circle at centre // line 1
  ```

  the `Str` constructor of `Str.of("Circle at centre")` is invoked.

-
  ```
  traced Str: (0.0, 0.0) // line 2
  traced map: Circle at centre (0.0, 0.0) // line 3
  traced flatMap: Circle at centre (0.0, 0.0) // line 4
  ```

  the first `join` method invokes `flatMap` which invokes `map` which in turn invokes the `Str` constructor of `this.centre.toStr()`.

-
  ```
  traced map: Circle at centre (0.0, 0.0) with radius 1.0 // line 5
  ```

  the second `join` method that takes in a `String` will invoke `map` directly.

Note that `trace` will end off with the same output as `print` after tracing the formation.

Here are examples with lazy evaluation.

```
jshell> Str.of(() -> { System.out.println("beep..."); return "$0 ==> ";}).
   ...> join(new Point(1.0, 1.0).toStr())
$.. ==> Str@...

jshell> Str.of(() -> { System.out.println("beep..."); return "$0 ==> ";}).
   ...> join(new Point(1.0, 1.0).toStr()).
   ...> trace()
beep...
traced Str: $0 ==>
traced Str: (1.0, 1.0)
traced map: $0 ==> (1.0, 1.0)
traced flatMap: $0 ==> (1.0, 1.0)
$0 ==> (1.0, 1.0)

jshell> Str.of(() -> { System.out.println("beep..."); return "$0 ==> ";}).
   ...> flatMap(x -> { System.out.println("ouch..."); return new Point(1.0, 1.0).toStr();})
$.. ==> Str@...

jshell> Str.of(() -> { System.out.println("beep..."); return "$0 ==> ";}).
   ...> flatMap(x -> { System.out.println("ouch..."); return new Point(1.0, 1.0).toStr();}).
   ...> trace()
beep...
traced Str: $0 ==>
ouch...
traced Str: (1.0, 1.0)
traced flatMap: (1.0, 1.0)
(1.0, 1.0)
```

Hint: Perform the traces within each `Supplier` that requires tracing.

Level 5

Depending on how `trace` is implemented above, you could have included `System.out.println(..)` statements throughout your code. Remove these *side-effects* by including an overloaded `trace` method that takes in an appropriate `Consumer`, so that the trace is "consumed" by this `Consumer` instead.

```
jshell> new Point(1.0, 1.0).toStr()
$.. ==> Str@...

jshell> new Point(1.0, 1.0).toStr().run(x -> System.out.println(x))
(1.0, 1.0)

jshell> new Point(1.0, 1.0).toStr().trace(x -> System.out.println(x))
traced Str: (1.0, 1.0)
(1.0, 1.0)

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr()
$.. ==> Str@...

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().run(x -> System.out.println(x))
Circle at centre (0.0, 0.0) with radius 1.0

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().trace(x -> System.out.println(x))
traced Str: Circle at centre
traced Str: (0.0, 0.0)
traced map: Circle at centre (0.0, 0.0)
traced flatMap: Circle at centre (0.0, 0.0)
traced map: Circle at centre (0.0, 0.0) with radius 1.0
```

```
Circle at centre (0.0, 0.0) with radius 1.0

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().run(x -> {})

jshell> new Circle(new Point(0.0, 0.0), 1.0).toStr().trace(x -> {})

jshell>
```

From the last two test cases, notice that the `Consumer` does nothing; hence there is no output.

More hints: modify how tracing is done within each `Supplier` in the level above. You may start by turning the traces on and off by letting `trace` take in a boolean argument, say `trace(true)` to trace. Then replace the `boolean` argument with a `Consumer` to consume the trace.

Level 6 (Ungraded)

Feel free to test out `Stringable::toStr()` and the `Str` methods `print`, `run`, `join`, `map`, `flatMap` and see if you can break "Tell-Don't-Ask" via string access and/or manipulation. You may write your tests in the file `level6.jsh`. Do keep in mind the premise that we will not allow students to define any methods that return `String`.

All contributions are welcome! Thank you very much.