In this quiz you will build and use an immutable abstract datatype `Weather`.

`Weather` describes the weather as a combination of a `temperature` and a `precipitation condition`. We might use it to represent, for example, the “average” weather for a day, or the weather at a particular point in time.

Provided at the bottom of the quiz page are:

- a skeleton for `Weather`, and
- an enumeration `Condition` to represent precipitation conditions, along with documentation for several useful `enum` methods.

You can open this text and the code in a separate tab by clicking the “open in separate tab” button to the right.

Problem 1. (28 points)

Design and implement `Weather`. Use the `Condition` enum. You must use the provided `rep` and all the code provided in this question and at the bottom of the page. Read all the parts of this question first, since they are interrelated and you may not wish to answer them in order!

For the specification comments, you do **not** need to use Javadoc syntax. Instead, write clear, useful, SFB, ETU, RFC single-line specs.

(a) Constants you wish to declare in `Weather`, if any. You may only declare constants, you may not declare additional `rep` fields.

```java
// none
```

(b) One-line spec comment for a creator that takes no arguments:

```java
create a weather description of 0 deg C and sunny
```

... and a constructor declaration & implementation to satisfy that spec:
As noted in the provided code, we will want other creators/producers, we're just not writing them now.

(c) One-line spec comment for observer condition, and its implementation:

```java
public Weather() {
    data = List.of(0, SUNNY.ordinal());
}
```

```java
public Condition condition() {
    return Condition.values()[data.get(1)];
}
```

(d) One-line spec comment for observer temperature, and its implementation:

```java
public int temperature() {
    return data.get(0);
}
```

(e) Abstraction function:

\[
AF(data) = \text{weather description of temperature data.get(0) deg C and precipitation condition the data.get(1)'th constant in Condition}
\]

(f) Rep invariant:

\[
data.size() == 2 \text{ (necessary since Weathers are equal iff data are equal)}
0 <= data.get(1) < Condition.values().length
\]

(g) Safety from rep exposure argument (ignore additional creators/producers we haven’t written yet):

- e.g.: data is private, final, an unmodifiable list, and contains immutable Integers

Problem 2. (18 points) random choice of part (a)

Suppose we intend to write a function:

```java
public static void sweaterWeather(List<Weather> days)
```

Global warming aside, hopefully there’s something about this spec you’d prefer to avoid. Refactor it to achieve the same purpose but be much more SFB/ETU. Don’t change the names sweaterWeather or days.

(a) In a phrase, what is the problem?

modifies the input, would be better to return a new List

Suppose we intend to write a function:
Global warming aside, hopefully there's something about this spec you'd prefer to avoid. Refactor it to achieve the same purpose but be much more ETU/RFC. Don't change the names skiingProbability or days.

(a) In a phrase, what is the problem?

operational, would be better to be declarative

(b) Write a revised one-line spec comment:

return a list identical to the input but including only the days that are CLOUDY or RAINY conditions and in the 50's Fahrenheit
-or- return the average of two probabilities: that a random day from days is SNOWY, and that a random day from days is below 32 Fahrenheit

... and method signature (if the signature is unchanged, you may write NO CHANGE):

public static List<Weather> sweaterWeather(List<Weather> days) -or- NO CHANGE

(c) To test this function, you’ll partition days.

Use the boxes below to write a single not-merely-correct-but-excellent 3-part partition (on days) that you would like to see in the test suite.

e.g. for sweaterWeather... days includes no elements that are either CLOUDY/RAINY or in the 50's Fahrenheit

days includes some elements that are either CLOUDY/RAINY and in the 50’s Fahrenheit, but none that are both

days includes some elements that are both CLOUDY/RAINY and in the 50’s Fahrenheit

Problem 3. (24 points) random choice of part (a)

Suppose we have this function:

// requires non-empty forecast mapping day-of-month numbers to forecasted weather
// returns a day number with a best available good-for-a-picnic forecast,
//   or -1 if there are no days good for a picnic
int picnicDay(Map<Integer, Weather> forecast);

For each option below:

- In the left box, write STRONGER, WEAKER, SAME, or INCOMPARABLE to say whether that new spec is stronger, weaker, the same as, or incomparable to the original spec above.
- In the right box, explain your answer by saying:
  - whether the precondition of the new spec is stronger, weaker, the same, or incomparable to the original above and why, and
  - whether the postcondition of the new spec is stronger, weaker, the same, or incomparable to the original and why.

(a) Write STRONGER, WEAKER, SAME, or INCOMPARABLE on the left; explain clearly on the right:

// requires non-empty forecast mapping day-of-month numbers to forecasted weather
// returns the earliest day number with a best available good-for-a-picnic forecast,
//   or -1 if there are no days good for a picnic
int picnicDay(Map<Integer, Weather> forecast);

STRONGER

the precondition is the SAME: it’s unchanged
the postcondition is STRONGER: the restriction "earliest" has been added

(a) Write STRONGER, WEAKER, SAME, or INCOMPARABLE on the left; explain clearly on the right:

// requires non-empty forecast mapping day-of-month numbers to forecasted weather
// returns a day number with a good-for-a-picnic forecast,
//   or -1 if there are no days good for a picnic
int picnicDay(Map<Integer, Weather> forecast);
Problem 4. (18 points) random choice of part (c)

Speaking of forecasts: Alyssa P. Hacker has developed a truly remarkable weather forecasting algorithm that she was able to fit in the margins of her copy of Structure and Interpretation of Computer Programs. Unfortunately, specs did not fit in the margins, but she has provided an example of how to use her Forecaster ADT:

```java
String today = "2020-10-19";
Weather sunny40 = ...;
Weather sunny42 = ...;
Forecaster crystalBall = new Forecaster();

// record some recent weather observations
crystalBall.record(today, "11:05", sunny40);
crystalBall.record(today, "11:10", sunny42);

// then forecast future weather one minute at a time
crystalBall.dial(today, "11:55");
Weather forecastElevenFiftyFive = crystalBall.forecast();
Weather forecastElevenFiftySix = crystalBall.forecast();
Weather forecastElevenFiftySeven = crystalBall.forecast();
// ...
```

(a) With respect to Forecaster, what kind(s) of ADT operation is `forecast()`? Write all that apply.
Ben Bitdiddle feels confident he can use Forecaster based on that example. He tries a few different things, but they don’t work.

(b) Immediately after running Alyssa’s example code above, Ben tries:

```java
crystalBall.record(today, "11:15", sunny42);
Weather w = crystalBall.forecast(); // <-- returns the forecast at 11:55, not 11:58!
```

Ben is surprised, but Alyssa is not. Hypothesize: what operation’s postcondition makes this behavior reasonable, and what is the relevant part of that postcondition? Make your hypothesis as straightforward as possible, keeping Forecaster as useful as possible.

The postcondition of record() says that the Forecaster returns to its dial()ed time for subsequent forecast()ing.

(c) Immediately after running Alyssa’s example code above, Ben tries:

```java
crystalBall.record(today, "11:00", sunny40);
Weather w = crystalBall.forecast(); // <-- throws ArrayIndexOutOfBoundsException!
```

Ben is surprised, but Alyssa is not. Hypothesize: what operation’s precondition makes this behavior reasonable, and what is the relevant part of that precondition? Make your hypothesis as straightforward as possible, keeping Forecaster as useful as possible.

The precondition of record() requires the time to be later than the latest record()ed observation. (Having violated that precondition, the Forecaster is broken; subsequent calls to forecast() fail.)

(c) Immediately after running Alyssa’s example code above, Ben tries:

```java
crystalBall.record(today, "11:10", sunny40);
Weather w = crystalBall.forecast(); // <-- throws ArithmeticException!
```

Ben is surprised, but Alyssa is not. Hypothesize: what operation’s precondition makes this behavior reasonable, and what is the relevant part of that precondition? Make your hypothesis as straightforward as possible, keeping Forecaster as useful as possible.

The precondition of record() does not permit multiple observations for the same time. (Having violated that precondition, the Forecaster is broken; subsequent calls to forecast() fail.)

(Note: for these questions, other hypotheses might be possible, if perhaps less straightforward or usefulness-preserving.)

Problem 5. (12 points)

Joey Coffeepun would rather work with immutable weather forecasts, so they write the following class that uses Alyssa’s Forecaster:

```java
/** Immutable weather forecast. */
class ImForecaster {

    private final Forecaster f;
    private final List<Weather> w;

    // requires date and time valid for Forecaster dialing
    // creates a forecast using observations recorded in f, starting at the given date and time
    public ImForecaster(Forecaster f, String date, String time) {
        this.f = f; // line 1
        this.f.dial(date, time); // line 2
        this.w = new ArrayList<>();
    }

    // returns forecasted weather for the given nonnegative number of minutes
    public List<Weather> getForecast(int minutes) {
        for (int i = w.size(); i < minutes; i++) {
            w.add(f.forecast());
        }
        return Collections.unmodifiableList(w.subList(0, minutes));
    }
}
```
In one sentence, why exactly is the first line of code in the constructor a problem?

Calling `f.dial()` mutates input `f`, which is not included in the spec and therefore not permitted.

And in one sentence, why exactly is the second line of code in the constructor a problem?

Assigning `this.f = f` puts an alias to mutable `f` from the client in the rep, creating rep exposure.

Leaving those serious problems aside, the code in `getForecast` does work! It relies on the rep invariant in order to achieve correctness.

Write a rep invariant for `ImForcaster` that is preserved by `getForecast` and sufficient to guarantee `getForecast`'s correctness.

(Hint: consider the contents of `w`, and consider the state of `f`.)

Weather

```java
/** Immutable description of weather. */
public class Weather {

  // you must use this rep:
  private final List<Integer> data;

  // ... some docs and operations to be completed in Problem 1 ...

  // ... other creators and/or producers we're not writing right now ...

  // you must use these implementations:
  @Override public boolean equals(Object that) {
      return that instanceof Weather && sameValue((Weather)that);
  }
  private boolean sameValue(Weather that) {
      return data.equals(that.data);
  }
  @Override public int hashCode() {
      return data.hashCode();
  }
}
```

Condition

```java
/** Precipitation conditions. */
public enum Condition {
    SUNNY, CLOUDY, RAINY, SNOWY
}
```

Note that Java automatically provides these static methods:

```java
/**
 * @return an array containing the constants of this enum type, in the order they're declared
 */
public static Condition[] values()

/**
 * @return the enum constant of this type with the specified name (must match exactly)
 * @throws IllegalArgumentException if this enum type has no constant with the specified name
 */
public static Condition valueOf(String name)
```

... and this instance method:

```java
/**
 * @return the ordinal position of this enumeration constant (its position in its enum declaration,
 * where the first constant is assigned an ordinal of zero)
 */
public int ordinal()
```