Exercise 1

Question 1.1

Suppose you're writing code that needs to turn the laundry machine on by calling the run method. There are at least four ways to accidentally call the run method incorrectly. (This is just counting single calls to run, not counting errors like calling run on a running machine.) What are they, and how would you prevent misuse?

Answer Times and temperatures can be negative; mode can be an invalid value. And the arguments may be given out of order.

Option 1: Distinct types for all parameters; each type restricts values.

```
enum Mode {
    LOW, MEDIUM, HIGH
}
```

```
public void run(Time time, Temperature temp, Mode mode)
```

Can call it like so:

```
run(Time.min(60), Temp.f(110), Mode.MEDIUM);
```

Option 2 (for order problem): Keyword arguments, or some simulacrum (e.g.: argument structure with the builder pattern)

```
class LaundryRunConfig {
    private final Time time;
    private final Temperature temp;
    private final Mode mode;
}
```

Question 1.2

The website should not be able to control the washing machines, nor access internal details about the state of the washer. We want to enforce this programmatically.

In the first design, LaundryDisplay has a direct reference to the list of WashingMachine's. How would you enforce that it can only access whether a laundry machine is on?

Answer Option 1

```
    public class WashingMachine implements RestrictedWashingMachine {
    ...
    public boolean isOn() { .... }
}
```

```
interface RestrictedWashingMachine { public boolean isOn(...) { ... }
 ↪ }

// LaundryDisplay just has a list of RestrictedWashingMachine's
```

Option 2

```
public class WashingMachine {
    ...

    public class RestrictedWashingMachine {
        public boolean isOn() {
            return WashingMachine.this.isOn();
        }
    }

    public RestrictedWachingMachine getRestrictedWashingMachine() {
        return new RestrictedWashingMachine();
    }

    public boolean isOn() { .... }

}

// LaundryDisplay has a list of RestrictedWashingMachine's
```

Question 1.3

In the second design, LaundryDisplay has a reference to the `Laundromat`, but not to the `WashingMachine's`. How would you enforce that it can only access whether a laundry machine is on?

Answer The most straightforward approach is to add a `boolean isWashingMachineOn(int index` to `Laundromat`. This however creates some hidden coupling with `WashingMachine` however: every time a new method is added to `WashingMachine`, if that new method is safe for public access, a corresponding method is likely to be added to `Laundromat`. So, an even better approach is to return the `WashingMachine` under a `RestrictedWashingMachine` interface so that no duplication is necessary.

```
public class WashingMachine implements RestrictedWashingMachine {
    ...
    public boolean isOn() { .... }
}

public class Laundromat {
    public RestrictedWashingMachine getWashingMachine(int i) {
        return washingMachines.get(i).getRestrictedWashingMachine()
```

```
    }
}
```

Exercise 2

Question 2.1

Design an API for a Tic-Tac-Toe board, consisting of types representing states of the board, along with functions move, takeMoveBack, whoWonOrDraw, and isPositionOccupied

```
enum GameResult {
  PLAYER_X,
  PLAYER_O,
  DRAW
}

class BoardCoordinates {
    // Enforced within range
    private int x;
    private int Y;
    public Position(int x, int y) {}
    // getters
}

// Game is sum type of StartingGame | InProgressGame | FinishedGame
// Use instanceof to distinguish them
interface Game {}
interface StartedGame extends Game {
    public UnfinishedGame takeMoveBack() {}
    public boolean isPositionOccupied(BoardCoordinates coordinates)
     ↪ {}
}
interface UnfinishedGame extends Game {
    public StartedGame move(BoardCoordinates coordinates) {}
}

class StartingGame implements UnfinishedGame {
    // Representation elided
}
class InProgressGame implements UnfinishedGame, StartedGame {
    // Representation elided
}
class FinishedGame implements StartedGame {
    // Representation elided
    public GameResult whoWonOrDraw() {}
}
```

Answer

Extra challenge 1

// no need to check since getPhotos only returns photos viewable for the context.def listPhotos(user, viewerContext): for photo is viewerContext.getPhotos(user, db): displayPhoto(photo)java

```
class GameResult(Enum):
  PLAYER_X = 1
  PLAYER_O = 2
  DRAW = 3

class BoardCoordinates:
  def __init__(self, x, y):
    # Enforce x and y range
    # Assign to self

class Game:
    def move(self, pos):
        raise NotImplementedError()
    def takeMoveBack(self):
        raise NotImplementedError()
    def whoWonOrDraw(self, context):
        raise NotImplementedError()
    def isPositionOccupied(self, pos):
        raise NotImplementedError()

class GameInitialState(Game):
    def move(self, coordinates):
        # Return GameStarted

class GameStarted(Game):
    def move(self, coordinates):
        # Return GameStarted or GameFinished;

    def takeMoveBack(self):
        # Return GameStarted or GameInitialState;

    def isPositionOccupied(self, coordinates):
        return True or False

class GameFinished(Game):
    def whoWonOrDraw(self, context):
        # Return the game result.
```

```
        def takeMoveBack(self):
            # Return GameStarted

        def isPositionOccupied(self, coordinates):
            return True or False
```

Answer

Extra challenge 2

Do this in a statically-typed language, but also make it a compile error if you try to play in a square that's already taken.

Answer Option 1: Generate distinct types for every board position, and distinct methods for every place.

Option 2: Use type parameters to give distinct types for every board position. E.g.:

```
public abstract class CS {} // CS = "CellState"
public final class O extends CellState {}
public final class X extends CellState {}
public final class E extends CellState {}

public class Board<P00 extends CS, P01 extends CS, P02 extends CS,
 ↪  P10 extends CS, P11 extends CS, P12 extends CS, P20 extends CS,
 ↪  P21 extends CS, P22 extends CS> {

    public static <P01 extends CS, ... , P22 extends CS Board<O, P01,
     ↪  P02, ..., P22> void playOInP00(Board<E, P01, ..., P22>) { ...
     ↪  }
// ...
}
```

Typescript version: https://tinyurl.com/2p89fer8

Haskell version:

```
{-# LANGUAGE DataKinds #-}
data CellState = E | O | X

-- The type variables are all phantom types
data Board (a :: CellState) (b :: CellState) <etc> (i :: CellState) =
 ↪  Board <list of cells>

isCell11Empty :: Board a b c d e f g h i -> Maybe (Board E b c d e f g
 ↪  h i)
isCell12Empty :: Board a b c d e f g h i -> Maybe (Board a E c d e f g
 ↪  h i)
<...>
```

```
placeOInCell11 :: Board E b c d e f g h i -> Board O b c d e f g h i
placeXInCell11 :: Board E b c d e f g h i -> Board X b c d e f g h i
placeOInCell12 :: Board a E c d e f g h i -> Board a O c d e f g h i
placeXInCell12 :: Board a E c d e f g h i -> Board a X c d e f g h i
<...>
```