Self-Service Course 4 Project - Kiva Robot Remote Control

About

Launch

Engage

Learn

Hire

Volunteer

FAQs

Contact

Current Participants

Internal

Project Overview

Welcome to 2012! *Ahem,* I mean to the Kiva team! We are working on introducing robots into the Fulfillment Centers and have plans to one day even have them drive themselves! But let's not get ahead of ourselves...

The Kiva robots are finished and they're incredible! They know how to move forward (F), turn left (L), turn right (R), take (T), and drop (D). What are they lifting and dropping? They are lifting and dropping pods filled with products and carrying them between the pod's storage location and a drop zone, where the Kiva robot remains, and a Fulfillment Center employee removes products from the pod to pack an order. We have created a program that prints out a map showing where the robot currently is, where the pod to pick up is, and where it needs to be dropped off.

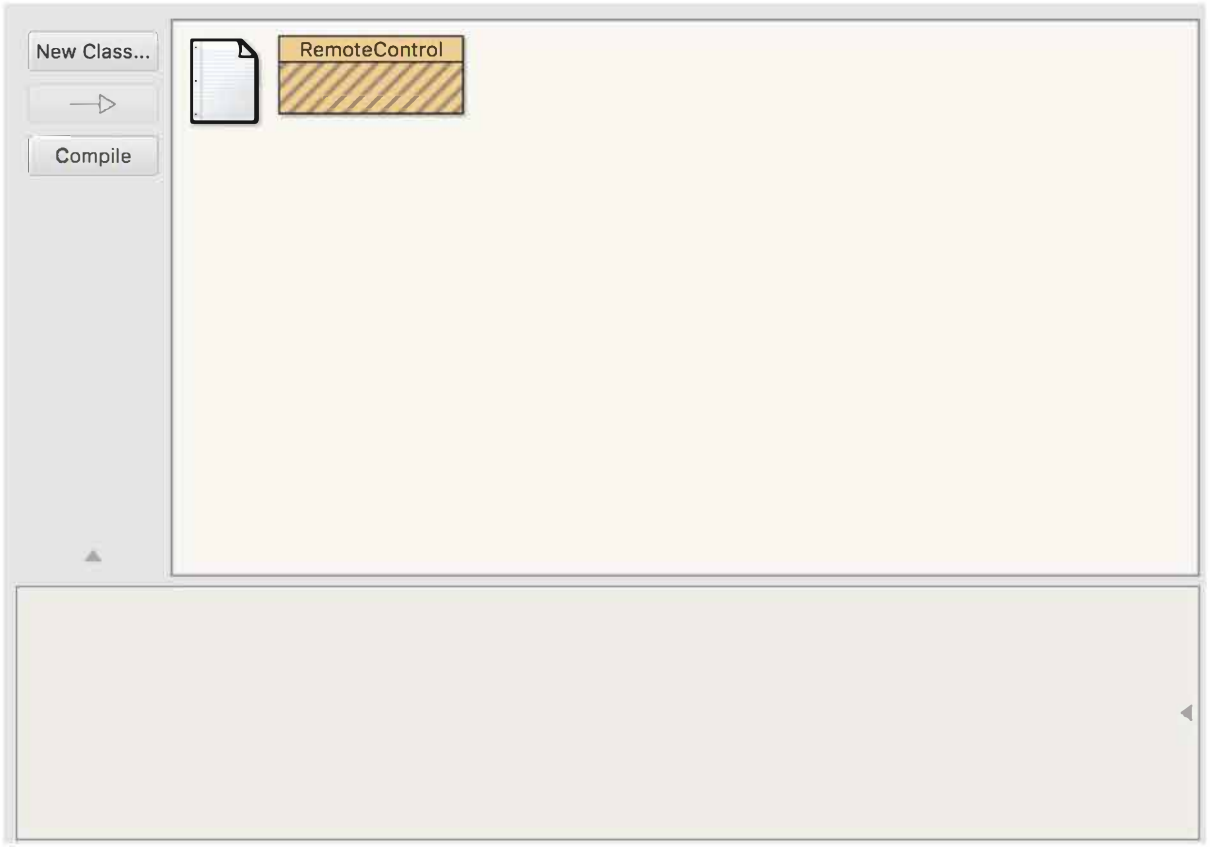
We need you to program a Kiva Robot Remote Control! Right now we've programmed the Remote Control to print out the map of the FC and then take in directions from the driver, but those directions aren't being sent to the robot. We need you to interpret the directions and send them to the robot. Kiva Robots are incredibly expensive to make so you need to make sure they won't drive into any walls or obstacles when you make a move.

P.S. We found this incredible video from the future showing “A Day in the Life of a Kiva Robot,” after they learned to drive themselves! https://www.youtube.com/watch?v=6KRjuuEVEZs

Before You Get Started

Download and Open the Project

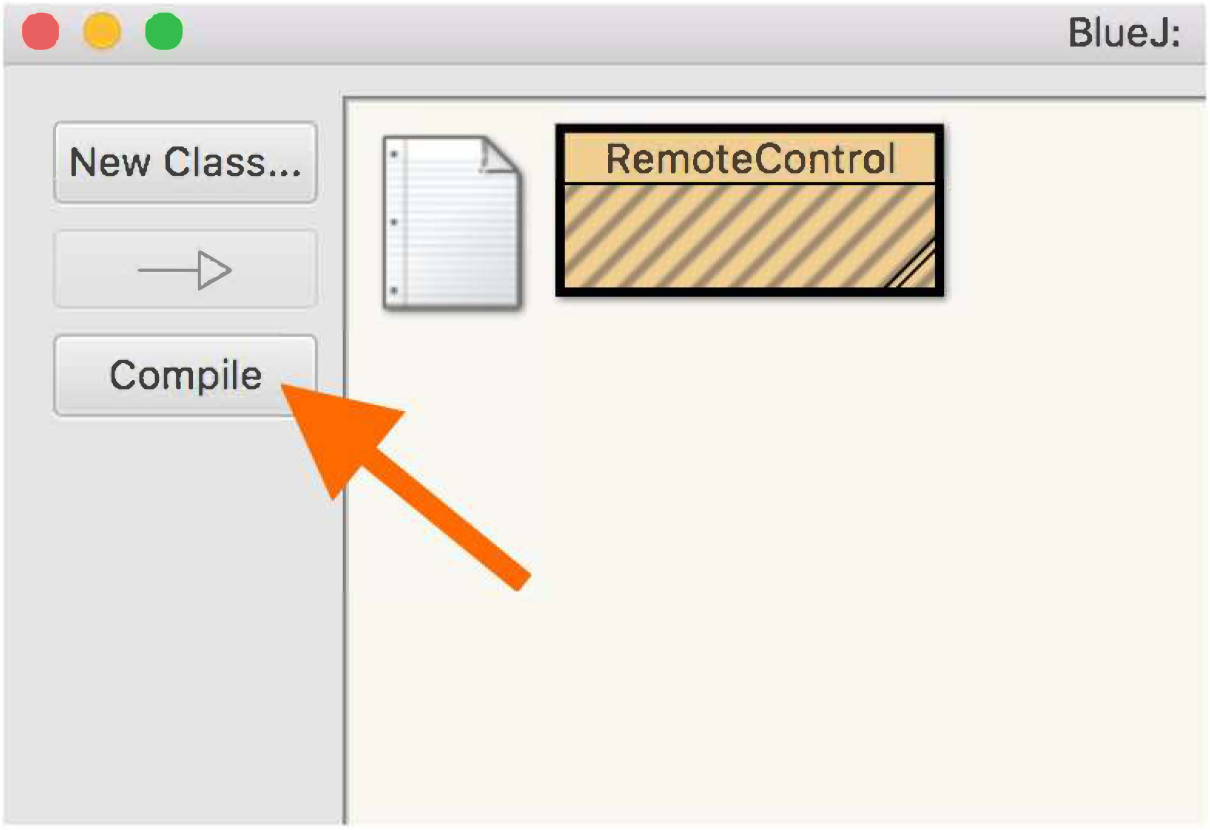
1. If you do not have BlueJ installed on your computer, please download it from Duke here . Download the appropriate version based on your operating system directly from that website. Don't download BlueJ directly from https://www.bluej.org/ because the version there will miss critical dependencies from Duke.
2. Open the KivaWorld downloads folder in workdocs.
3. Download the **KivaWorld-##.zip** file (for example, KivaWorld-01.zip, but may have a different number in place of the **"##"**) that you see
    1. There should be a **KivaWorld-##.zip** file that contains the starter code -- download this one now
    2. (There will also be a KivaWorldJavadoc-##.zip file that contains documentation about the existing code -- more on this below in a "Documentation" section)
4. Double-click on the .zip file to expand the project contents. Feel free to move the KivaWorld folder to wherever you keep your projects on your computer.
5. Double-click on KivaWorld/package.bluej. (Note that this is just another way of opening a BlueJ project--besides opening BlueJ first and using the Project menu to find the project)
6. You should see the `RemoteControl` class that you'll be implementing, and can now run the code!



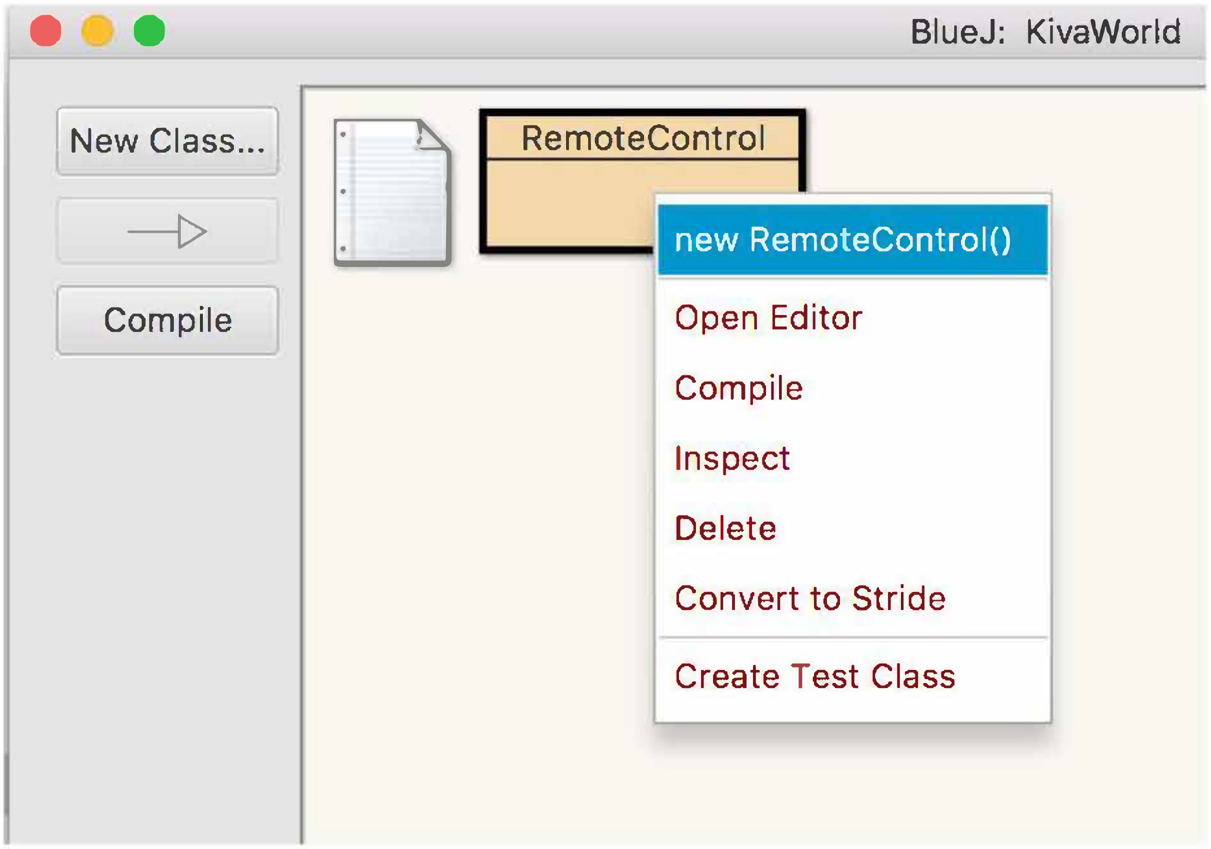
How to Run and Test the Project

Now that you've got the project open in BlueJ, let's run the code for the first time!

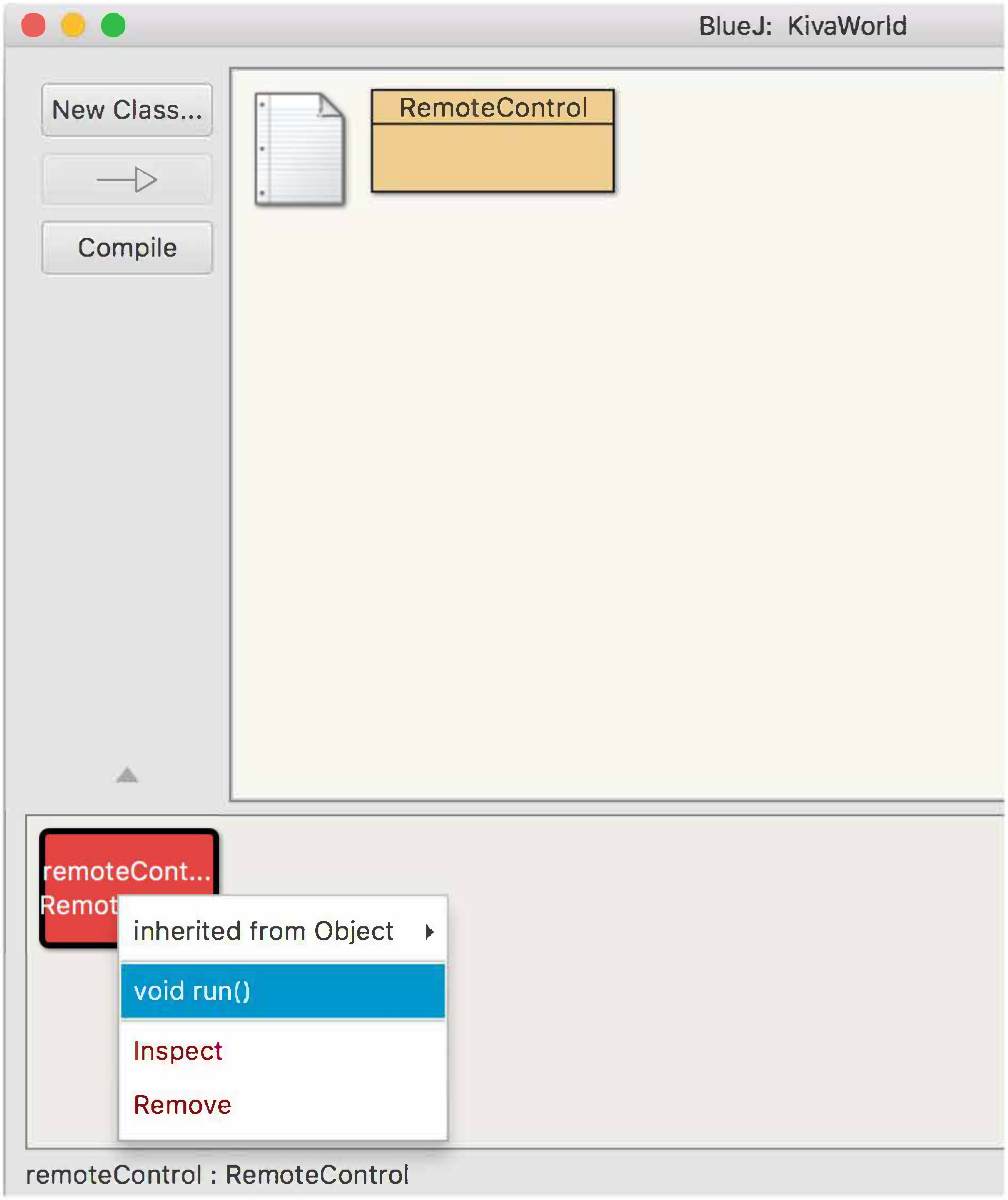
- Click “Compile” on the left side of the BlueJ project window:



- Right click on the `RemoteControl` class and select “new RemoteControl()” to create a new `RemoteControl` instance:



- Choose a name for your new instance, for example, remoteControl, and hit OK.
- Right-click on the red instance in the lower portion of the BlueJ window and select “void run()” to execute `RemoteControl`'s run instance method:



- The console should appear and you will get a `NullPointerException` at this point, which is expected. Don't worry, we'll get around to fixing that later! Now you're ready to get started!

What is Already Present in the Project?

The `RemoteControl` class is responsible for running the program, interacting with the user, and moving the Kiva robot. We have provided you this class with some starter code in it. It contains logic to interact with the console by printing out the map of the floor, and taking in the string of directions from the user.

In addition to `RemoteControl`, the project will utilize some code that has already been written by the Warehouse Team. The Warehouse Team had already written this code, so they've packaged it up for our team to use. You will not need to make changes to any of these classes, but we wanted to explain how this works so it doesn't seem like too much magic. These classes are tucked away inside a JAR (**J**ava **AR**chive) file inside KivaWorld/+libs/. A JAR file is a convenient way to bundle a bunch of related Java classes (often called a *library*) into a single file that the JVM can access. (*Optional: Here is a good intro on JARs , though a little more information than you need right now.*)

Here's a quick summary of the classes and enumerations provided by the Warehouse Team:

- `FloorMap`: Represents a Kiva's environment, including pickup/dropoff points, obstacles and Kiva's starting location. It allows identifying what exists at any given location, as well as rendering the entire map as a String for display.
- `FacingDirection`: Represents the direction a robot is facing. There are four orientations, and each includes a “delta” point which moves the robot one space in that direction when added to the current location.
- `FloorMapObject`: Represents what is located at any given location of the FloorMap.

Optional: Taking a look inside the JAR

Hide
If you'd like to see the JAR's contents, open a terminal window, cd into the project directory (see “How to open a terminal” below), and type jar -tvf +libs/*.jar. You should see a few .class files that correspond to the classes below! The .class file contents will be completely unintelligible to you, because they're bytecode, binary format that JVMs can understand but humans cannot.)

Optional: How to open a terminal and cd to the project directory

Hide
In Windows: Hit the Windows key, type “cmd” and hit Enter and you should have a window with a black background.

In MacOS: In Finder, select Applications → Utilities → Terminal.

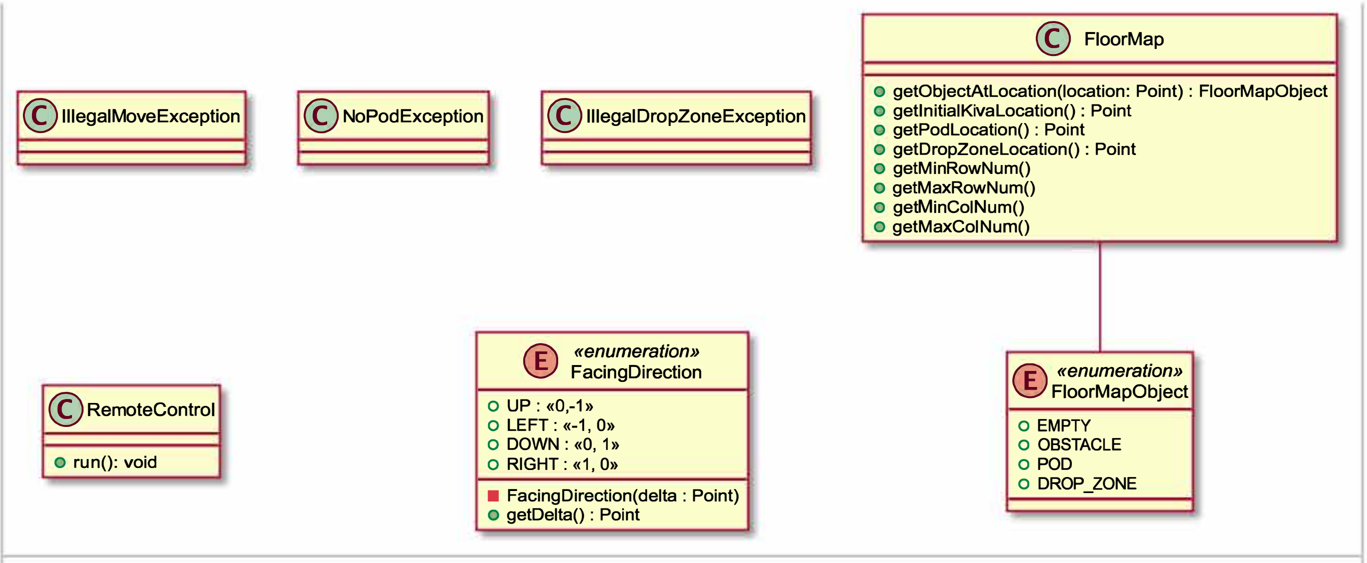
In both cases, let's move to the KivaWorld directory. This is where the +libs directory is. Type cd and then the full path of the directory containing the project.
For example: cd /Users/userName/workplace/KivaWorld

Starter Code Class Diagram

If you downloaded the project before 04/29/2021 and were trying to use the `FloorMap` methods `getPodLocation()` and `getDropZoneLocation()` but couldn't find them, you can update your project to get those methods by...

1. Downloading the latest version of `KivaWorldJavadoc-01.zip` from Shared Curriculum/KivaWorld , replacing the version of the javadoc you downloaded originally
2. Downloading the latest version of `KivaWorld-01.zip` from Shared Curriculum/KivaWorld and...
    1. Uncompressing `KivaWorld-01.zip`.
    2. Replacing the file `KivaWorld/+lib/KivaWorld-1.0.jar` in your copy of the project with the new version.

We created a class diagram to visualize the classes provided by the Warehouse Team and the `RemoteControl` class.



Actions: Pop Out | Edit | ?

Documentation

We've also provided more complete documentation in the form of Javadocs, which are a collection of HTML files that document classes and their methods.

1. Open the KivaWorld downloads folder on Workdocs.
2. Download the KivaWorldJavadoc-##.zip file with the highest version number (the version should match the version of the KivaWorld-##.zip file you downloaded, let us know if this is not the case).
3. Double-click on **index.html** to load the javadoc viewer in your browser.
4. Refer to these pages throughout the project to see how the above classes behave.

You can view the Javadocs for the classes you're writing and editing (e.g. `RemoteControl`) as you build them, via the Tools → Project Documentation menu in BlueJ (if it asks you if you want to regenerate documentation, click “Regenerate”).

Sample Floor Map Files

We have also provided you with three floor map files in the KivaWorld-##.zip that you downloaded. You can use these for testing purposes.

- sample_floor_map1.txt - a simple map with walls, but no obstacles. The K, P, and D are all in a straight line.
- sample_floor_map2.txt - a map with walls and obstacles
- sample_floor_map3.txt - a map with walls and obstacles

Project Goals

Your goal is to complete all of the project tasks, which will result in `RemoteControl` interacting with the user in the following way.

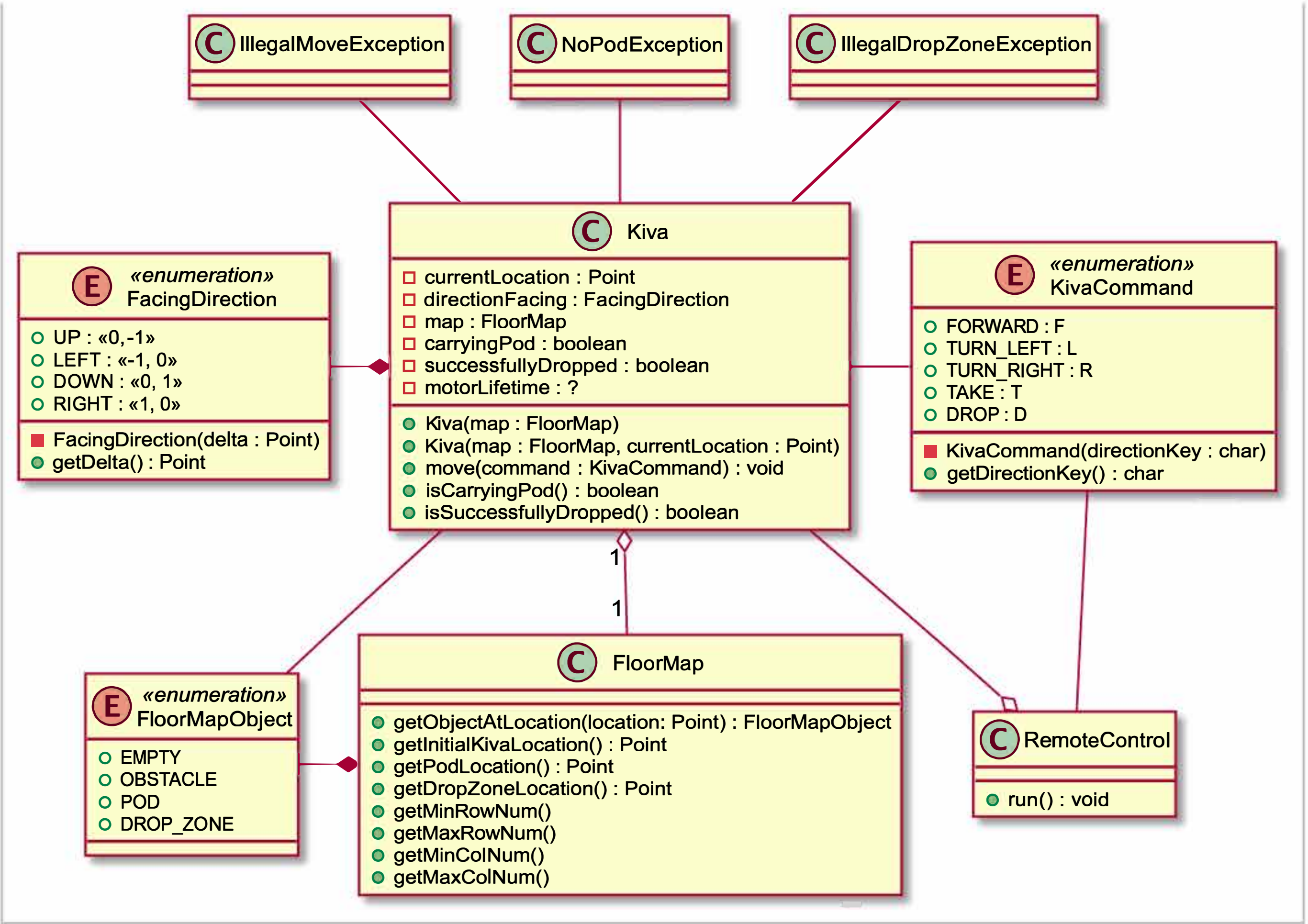
```
Please select a map file.

----------------
|        P    *|
|   **        *|
|   **        *|
| *K       D *|
---------------

Current Kiva Robot location: (3,4)
Facing: UP
Please enter the directions for the Kiva Robot to take.
FFFRFFFFFFTFFRFFFD


Successfully picked up the pod and dropped it off. Thank you!
```

The user will not always provide a path that successfully picks up and drops off the pod. We will detail out these error cases later and how to handle them, but as an example, here the path provided never picks up the pod.

```
Please select a map file.

 -------------|
         P   *|
    ---      -|
    |*|      *|
   K---    D -|
  *     * *  *|
 -------------|

Current Kiva Robot location: (2,4)
Facing: UP
Please enter the directions for the Kiva Robot to take.
FLFFFFFF

I'm sorry. The Kiva Robot did not pick up the pod and then drop it off in the right
place.
```

End State Class Diagram

This diagram shows you what your project will look like once you have completed all of the project tasks. Take a look at the new class and enum that you will be responsible for implementing, `Kiva` and `KivaCommand`. `RemoteControl` will interact with both of these new classes, and your `Kiva` class will use the already defined classes from the Warehouse Team. We will dive into more details in the implementation overview below.


IllegalMoveException
NoPodException
IllegalDropZoneException
Kiva
currentLocation : Point
directionFacing : FacingDirection
map : FloorMap
carryingPod : boolean
successfullyDropped : boolean
motorLifetime : ?
Kiva(map : FloorMap)
Kiva(map : FloorMap, currentLocation : Point)
move(command : KivaCommand) : void
isCarryingPod() : boolean
isSuccessfullyDropped() : boolean
«enumeration» FacingDirection
UP : «0,-1»
LEFT : «-1, 0»
DOWN : «0, 1»
RIGHT : «1, 0»
FacingDirection(delta : Point)
getDelta() : Point
«enumeration» KivaCommand
FORWARD : F
TURN_LEFT : L
TURN_RIGHT : R
TAKE : T
DROP : D
KivaCommand(directionKey : char)
getDirectionKey() : char
1
1
«enumeration» FloorMapObject
EMPTY
OBSTACLE
POD
DROP_ZONE
FloorMap
getObjectAtLocation(location: Point) : FloorMapObject
getInitialKivaLocation() : Point
getPodLocation() : Point
getDropZoneLocation() : Point
getMinRowNum()
getMaxRowNum()
getMinColNum()
getMaxColNum()
RemoteControl
run() : void

Actions: Pop Out | Edit | ?

Implementation Overview

KivaCommand describes the “moves” a Kiva robot can make: turning left or right, moving forwards, taking a pod, and dropping a pod. It also includes each move's one-letter abbreviation.

Kiva holds the state and logic of a robot: where it is, which way it's facing, whether it's carrying a pod right now, and whether it has successfully dropped the pod. A Kiva always starts facing UP (one of our provided FacingDirections), not carrying a pod, and not having successfully dropped the pod. The move() method performs the logic for any KivaCommand, and the isSuccessfulDrop() method indicates whether the previous command dropped the pod on the drop zone. We normally don't add getters to class diagrams, but this is the first time you're encountering a common naming convention that ATA will follow: boolean getters often follow the pattern “ isFieldName() ” or “ hasFieldName() ”.

When an impossible move is attempted, move() throws an exception. IllegalMoveException is appropriate for attempts to move the robot off the warehouse floor or into an obstacle; NoPodException is expected when there's an attempt to TAKE at a location without a POD; and IllegalDropZoneException should occur when we try to drop a pod anywhere but a DROP_ZONE. These exceptions are provided to you by the Warehouse Team.

Running the whole show is RemoteControl, initiated by executing its run() method. RemoteControl prompts the user to select a text file containing a FloorMap to load, prompts the user to input a chain of KivaCommands using single-letter abbreviations, and then executes the KivaCommands until Kiva drops the pod successfully into the drop zone (or encounters an error).

Now you're ready to get started. We've broken out what you need to do to fully program the remote control into the project tasks below. Happy coding!

Project Tasks

1. Implement the KivaCommand Enum

Before starting this task please refer back to the Enums Learning Material as a refresher.

Kiva objects can perform actions such as moving around and picking and dropping items. We allow users in our program to enter 1 letter representations of these moves ('F', 'R', 'L', 'T', or 'D'), so we need a way to map each action to a particular keyboard key.

Referring to the class diagram above, you'll do this by creating a new KivaCommand enum for the actions that the robot can perform. The enum will use values that correspond to particular keyboard characters. That way the RemoteControl class can map a char input key to a KivaCommand enum value.

In order to create an enum in BlueJ, click the 'New Class...' button. Enter the name of your enum, leave the Class Language as Java, and then select 'Enum' from the options for Class Type. Click 'OK' to create your enum.

Test this by creating a KivaCommandTester class. Create one test method per command and test that when you run each method you get the expected output. Here is an example KivaCommandTester class with one method filled in:

```java
public class KivaCommandTester {
    public void testForward() {
        KivaCommand command = KivaCommand.FORWARD;
        System.out.println(command);
        System.out.println(command.getDirectionKey());
    }

    // For you: create four more methods, each testing a different command.
}
```

Test Cases

Test: testForward()
Input: Creates a KivaCommand.FORWARD object, then prints the command and its direction key
Output: “FORWARD” followed by “F” is printed out to the console.

Test: testTurnLeft()
Input: Creates a KivaCommand.TURN_LEFT object, then prints the command and its direction key
Output: “TURN_LEFT” followed by “L” is printed out to the console.

Test: `testTurnRight()`
Input: Creates a `KivaCommand.TURN_RIGHT` object, then prints the command and its direction key
Output: “TURN_RIGHT” followed by “R” is printed out to the console.

Test: `testTake()`
Input: Creates a `KivaCommand.TAKE` object, then prints the command and its direction key
Output: “TAKE” followed by “T” is printed out to the console.

Test: `testDrop()`
Input: Creates a `KivaCommand.DROP` object, then prints the command and its direction key
Output: “DROP” followed by “D” is printed out to the console.

2. Create the `Kiva` Class

Refer to the Class Diagrams learning materials to refresh yourself on reading class diagrams. Next, we'll implement the `Kiva` class from the class diagram above. This is a fairly large and complicated task, so we'll break it into smaller steps.

Implement the Attributes, Getters, and Constructors

We will start by creating the class, adding its instance variables, and writing the getter methods and constructors. Note there are two constructors for this class, and you must implement both of them. One of them instantiates a `Kiva` object and sets the location of the Kiva robot from the provided `FloorMap`; the other instantiates a `Kiva` object and sets the location of the Kiva robot from a `Point` (remember to import `edu.duke.Point`). Both constructors should initialize the state with the robot facing up, not carrying any pod, and not having successfully dropped a pod. Ignore `motorLifetime` for now; we'll cover that later. Remember the getters for booleans follow the naming convention described above, `isSuccessfullyDropped()` and `isCarryingPod()`.

Test the `Kiva` constructors by creating a `KivaConstructorTest` class. First, we will create a `FloorMap` that we can pass to our `Kiva` constructors. We can do this by using the provided `FloorMap(String map)` constructor and the `defaultLayout` `String` in the example below. Create a test method for each `Kiva` constructor, using the same test `FloorMap` for each, and checking that the resulting Kiva robot location matches the input. (We don't write separate tests for “trivial” getters that contain no logic.) Here's the test for the constructor using a single argument:

```
import edu.duke.Point;

public class KivaConstructorTest {
    String defaultLayout = ""
                            + "-------------\n"
                            + "        P   *\n"
                            + "   **       *\n"
                            + "   **       *\n"
                            + "  K       D *\n"
                            + " * * * * * **\n"
                            + "-------------\n";
    FloorMap defaultMap = new FloorMap(defaultLayout);

    public void testSingleArgumentConstructor() {
        // GIVEN
        // The default map we defined earlier

        // WHEN
        // We create a Kiva with the single-argument constructor
        Kiva kiva = new Kiva(defaultMap);

        // THEN
        // The initial Kiva location is (2, 4)
        Point initialLocation = kiva.getCurrentLocation();
        Point expectedLocation = new Point(2, 4);
        if (sameLocation(initialLocation, expectedLocation)) {
            System.out.println("testSingleArgumentConstructor SUCCESS");
        } else {
            System.out.println(String.format( "testSingleArgumentConstructor FAIL:
%s != (2,4)!", initialLocation));
        }
    }

    private boolean sameLocation(Point a, Point b) {
        return a.getX() == b.getX() && a.getY() == b.getY();
    }

    // For you: create a test for the constructor taking two arguments.
}
```

Test Cases

Test: testSingleArgumentConstructor()
Input: Pass the constructor the default map as the argument, then validate that the currentLocation of the Kiva is as expected.
Output: "testSingleArgumentConstructor SUCCESS" is printed to the console.

Test: testTwoArgumentConstructor()
Input: Pass the constructor the default map and a location of (5, 6) as the arguments, then validate that the currentLocation of the Kiva is as expected.
Output: "testTwoArgumentConstructor SUCCESS" is printed to the console.

3. Implement Kiva's Move Method

Moving Forward

Next consider the `move()` method. Depending on the `KivaCommand` it receives, `move()` must update the current location, the direction the robot is pointing, whether it is carrying a pod or if the pod has been successfully dropped. This is a lot of logic; implementing a helper method for each command will make the code more readable and easier to manage. For now ignore the motorLifetime field.

Write a void helper method for the `FORWARD` command (a name such a `moveForward()` would be appropriate, but you may use any name you like). Use only the simplest logic, without regard for obstacles, pods, drop zones, or whether the robot is actually carrying anything; simply assume that the command is valid and update the state accordingly. This command should update the current location. When the move method gets a `KivaCommand` that is equal to `FORWARD` you should call your new helper method.

Test `move()` by creating a `KivaMoveTest` class. Create a test method for the `FORWARD` command and test that after you issue the command, *all* of the getters return the expected values. Here's a free example, `testForwardFromUp()`, with a bonus helper method that validates the `Kiva`'s state. You can use it for your other tests.

```
import edu.duke.Point;

public class KivaMoveTest {
    // Define the FloorMap we'll use for all the tests
    String defaultLayout = ""
                           + "-------------\n"
                           + "        P   *\n"
                           + "   **       *\n"
                           + "   **       *\n"
                           + "  K       D *\n"
                           + " * * * * * **\n"
                           + "-------------\n";

    FloorMap defaultMap = new FloorMap(defaultLayout);

    public void testForwardFromUp() {
        // GIVEN
        // A Kiva built with the default map we defined earlier
        Kiva kiva = new Kiva(defaultMap);

        // WHEN
        // We move one space forward
        kiva.move(KivaCommand.FORWARD);

        // THEN
        // The Kiva has moved one space up
        verifyKivaState("testForwardFromUp",
            kiva, new Point(2, 3), FacingDirection.UP, false, false);
    }

    // For you: create all the other tests and call verifyKivaState() for each

    private boolean sameLocation(Point a, Point b) {
        return a.getX() == b.getX() && a.getY() == b.getY();
    }

    private void verifyKivaState(
            String testName,
            Kiva actual,
            Point expectLocation,
            FacingDirection expectDirection,
            boolean expectCarry,
            boolean expectDropped) {
```

```
        Point actualLocation = actual.getCurrentLocation();
        if (sameLocation(actualLocation, expectLocation)) {
            System.out.println(
                    String.format("%s: current location SUCCESS", testName));
        }
        else {
            System.out.println(
                    String.format("%s: current location FAIL!", testName));
            System.out.println(
                    String.format("Expected %s, got %s",
                            expectLocation, actualLocation));
        }

        FacingDirection actualDirection = actual.getDirectionFacing();
        if (actualDirection == expectDirection) {
            System.out.println(
                    String.format("%s: facing direction SUCCESS", testName));
        }
        else {
            System.out.println(
                    String.format("%s: facing direction FAIL!", testName));
            System.out.println(
                    String.format("Expected %s, got %s",
                            expectDirection, actualDirection));
        }

        boolean actualCarry = actual.isCarryingPod();
        if (actualCarry == expectCarry) {
            System.out.println(
                    String.format("%s: carrying pod SUCCESS", testName));
        }
        else {
            System.out.println(
                    String.format("%s: carrying pod FAIL!", testName));
            System.out.println(
                    String.format("Expected %s, got %s",
                            expectCarry, actualCarry));
        }

        boolean actualDropped = actual.isSuccessfullyDropped();
        if (actualDropped == expectDropped) {
            System.out.println(
                    String.format("%s: successfully dropped SUCCESS", testName));
        }
        else {
            System.out.println(
                    String.format("%s: successfully dropped FAIL!", testName));
            System.out.println(
                    String.format("Expected %s, got %s",
                            expectDropped, actualDropped));
        }
    }
}
```

Moving Forward Test Cases

Test: `testForwardFromUp()`
Input: Create a `Kiva` object using the default map, and call move with the `FORWARD` command. Verify the state of the `Kiva` object. Ensure the current location has changed.
Output: The following is printed to the console:

```
testForwardFromUp: current location SUCCESS
testForwardFromUp: facing direction SUCCESS
testForwardFromUp: carrying pod SUCCESS
testForwardFromUp: successfully dropped SUCCESS
```

Turning Left

Continue implementing and testing `move()` with the next command, `TURN_LEFT`. Once you have implemented its helper method, write tests for it in the `KivaMoveTest` class. Copy the existing test method to create one test for each possible case of the directionFacing point (`UP`, `DOWN`, `LEFT`, and `RIGHT`). Ensure that each test verifies the `Kiva`'s state.

Turning Left Test Cases

Test: `testTurnLeftFromUp()`
Input: Create a `Kiva` object using the default map, and call move with the `TURN_LEFT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `LEFT`.
Output: The following is printed to the console:

```
testTurnLeftFromUp: current location SUCCESS
testTurnLeftFromUp: facing direction SUCCESS
testTurnLeftFromUp: carrying pod SUCCESS
testTurnLeftFromUp: successfully dropped SUCCESS
```

Test: `testTurnLeftFromLeft()`
Input: Create a `Kiva` object using the default map, and call move twice with the `TURN_LEFT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `DOWN`.
Output: The following is printed to the console:

```
testTurnLeftFromLeft: current location SUCCESS
testTurnLeftFromLeft: facing direction SUCCESS
testTurnLeftFromLeft: carrying pod SUCCESS
testTurnLeftFromLeft: successfully dropped SUCCESS
```

Test: `testTurnLeftFromDown()`
Input: Create a `Kiva` object using the default map, and call move three times with the `TURN_LEFT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `RIGHT`.
Output: The following is printed to the console:

```
testTurnLeftFromDown: current location SUCCESS
testTurnLeftFromDown: facing direction SUCCESS
testTurnLeftFromDown: carrying pod SUCCESS
testTurnLeftFromDown: successfully dropped SUCCESS
```

Test: `testTurnLeftFromRight()`
Input: Create a `Kiva` object using the default map, and call move four times with the `TURN_LEFT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `UP`.
Output: The following is printed to the console:

```
testTurnLeftFromRight: current location SUCCESS
testTurnLeftFromRight: facing direction SUCCESS
testTurnLeftFromRight: carrying pod SUCCESS
```

```
testTurnLeftFromRight: successfully dropped SUCCESS
```

Food For Thought: We had to make a lot of if-then choices to determine which direction resulted from a left turn. The `FacingDirection` enum is provided, and cannot be changed; if we *could* change it, could we add functionality to know which direction was "left of" any other direction? Why would we want to?

Updating Moving Forward Test Cases

The existing tests for `FORWARD` only covers the case where the robot is facing `UP`. Implementing `TURN_LEFT` allows us to check moving forward from any of four facing directions.

Test the new starting conditions when executing the `FORWARD` command. Copy the test method for `FORWARD` and modify it to account for each possibility. Make each test independent: don't repeat a "turn and check" three times. Instead, write one test that constructs a `Kiva`, turns left, moves forward, and checks all the state; then write a test that constructs a `Kiva`, turns left twice, moves forward, and checks all the state; and so on.

Test: `testForwardWhileFacingLeft()`
Input: Create a `Kiva` object using the default map, and call move once to `TURN_LEFT`, and again to move `FORWARD`. Verify the state of the Kiva object. Ensure the direction facing is now `LEFT`, and the `currentLocation` has updated to one space to the left.
Output: The following is printed to the console:

```
testForwardWhileFacingLeft: current location SUCCESS
testForwardWhileFacingLeft: facing direction SUCCESS
testForwardWhileFacingLeft: carrying pod SUCCESS
testForwardWhileFacingLeft: successfully dropped SUCCESS
```

Test: `testForwardWhileFacingDown()`
Input: Create a `Kiva` object using the default map, and call move twice with the `TURN_LEFT` command, and again to move `FORWARD`. Verify the state of the `Kiva` object. Ensure the direction facing is now `DOWN`, and the `currentLocation` has updated to one space below.
Output: The following is printed to the console:

```
testForwardWhileFacingDown: current location SUCCESS
testForwardWhileFacingDown: facing direction SUCCESS
testForwardWhileFacingDown: carrying pod SUCCESS
testForwardWhileFacingDown: successfully dropped SUCCESS
```

Test: `testForwardWhileFacingRight()`
Input: Create a `Kiva` object using the default map, and call move three times with the `TURN_LEFT` command, and again to move `FORWARD`. Verify the state of the `Kiva` object. Ensure the direction facing is now `RIGHT`, and the `currentLocation` has updated to one space to the right.
Output: The following is printed to the console:

```
testForwardWhileFacingRight: current location SUCCESS
testForwardWhileFacingRight: facing direction SUCCESS
testForwardWhileFacingRight: carrying pod SUCCESS
testForwardWhileFacingRight: successfully dropped SUCCESS
```

Implement Turning Right, Taking, and Dropping

Continue implementing and testing move() one command at a time, updating the state variables as if the command always succeeds. As each command is implemented, add tests for it in the `KivaMoveTest` class.

Note that it is possible to check all combinations of outcomes for each command. For instance, you could test that `TAKE` works for *each* direction. The number of conditions is small enough to make this possible, but large enough to be extremely cumbersome. This happens in actual development, too.

To reduce the burden, we will often examine the decisions that must be made and only write tests checking each condition that leads to a decision. For instance, since the code for `TURN_LEFT` must make decisions accounting for the four possible conditions of the `directionFacing` field, we should have four tests, one test for each condition. (We should also have a test for the fifth condition, when `directionFacing` is null, but we'll learn how to do that in a later unit.) On the other hand, the `TAKE` logic has no conditions, so we only need one test.

Turning Right, Taking, and Dropping Test Cases

Test: `testTurnRightFromUp()`
Input: Create a `Kiva` object using the default map, and call `move()` with the `TURN_RIGHT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `RIGHT`.
Output: The following is printed to the console:

```
testTurnRightFromUp: current location SUCCESS
testTurnRightFromUp: facing direction SUCCESS
testTurnRightFromUp: carrying pod SUCCESS
testTurnRightFromUp: successfully dropped SUCCESS
```

Test: `testTurnRightFromLeft()`
Input: Create a `Kiva` object using the default map, and call `move()` with the `TURN_LEFT` command followed by a call to move with the `TURN_RIGHT` command. Verify the state of the `Kiva` object. Ensure the direction facing is now `UP`.
Output: The following is printed to the console:

```
testTurnRightFromLeft: current location SUCCESS
testTurnRightFromLeft: facing direction SUCCESS
testTurnRightFromLeft: carrying pod SUCCESS
testTurnRightFromLeft: successfully dropped SUCCESS
```

Test: `testTurnRightFromDown()`
Input: Create a `Kiva` object using the default map, and call `move()` with the `TURN_LEFT` command twice followed by a call to move with the TURN_RIGHT command. Verify the state of the `Kiva` object. Ensure the direction facing is now `LEFT`.
Output: The following is printed to the console:

```
testTurnRightFromDown: current location SUCCESS
testTurnRightFromDown: facing direction SUCCESS
testTurnRightFromDown: carrying pod SUCCESS
testTurnRightFromDown: successfully dropped SUCCESS
```

Test: `testTurnRightFromRight()`
Input: Create a `Kiva` object using the default map, and call `move` with the `TURN_LEFT` command three times followed by a call to move with the `TURN_RIGHT` command. Verify the state of the `Kiva` object.

Ensure the direction facing is now `DOWN`.
Output: The following is printed to the console:

```
testTurnRightFromRight: current location SUCCESS
testTurnRightFromRight: facing direction SUCCESS
testTurnRightFromRight: carrying pod SUCCESS
testTurnRightFromRight: successfully dropped SUCCESS
```

Test: `testTakeOnPod()`
Input: Create a `Kiva` object using the default map, and call `move()` to go up three times, turn right, move right six times, and take the pod. Verify the state of the `Kiva` object. Ensure that it is carrying the pod.
Output: The following is printed to the console:

```
testTakeOnPod: current location SUCCESS
testTakeOnPod: facing direction SUCCESS
testTakeOnPod: carrying pod SUCCESS
testTakeOnPod: successfully dropped SUCCESS
```

Test: `testDropOnDropZone()`
Input: Create a `Kiva` object using the default map, and call `move()` to go up three times, turn right, move right six times, take the pod, move to the drop zone, and drop the pod. Verify the state of the `Kiva` object. Ensure that it is not carrying the pod, and the drop was successful.
Output: The following is printed to the console:

```
testDropOnDropZone: current location SUCCESS
testDropOnDropZone: facing direction SUCCESS
testDropOnDropZone: carrying pod SUCCESS
testDropOnDropZone: successfully dropped SUCCESS
```

Now that we have tests that verify all the possible `Kiva` moves, we can improve our code and run these tests whenever we want to verify that we haven't broken anything.

4. Handle Invalid Moves

Before starting this task please refer back to the Throwing Exceptions Learning Material as a refresher.

We've been so preoccupied with whether or not we *could* that we didn't stop to think if we *should*. Let's improve the `move()` method to detect error conditions. We'll modify our helper methods so that if the requested move isn't possible, we'll throw an `IllegalMoveException`. When attempting to take a pod on a space where no pod exists, we'll throw a `NoPodException`. When attempting to drop at any space other than a drop zone, we'll throw an `IllegalDropZoneException`.

We'll use the `FloorMap`'s `getObjectAtLocation()`, `getMaxRowNum()`, and `getMaxColNum()` methods to check whether the move is legal. Turning should always be legal, moving forward is only legal if the new location is on the floor and there isn't anything blocking. Again, we'll break these into single steps and write tests as we go.

Implement bounds checking in the helper method for moving forward. Before changing the current location, calculate where the new location *would* be if the move is successful. If the new location is less than 0 in any dimension, or greater than the corresponding dimension of the floor map, throw an `IllegalMoveException` with a message that describes the problem.

```
for (let i = 0, len = cars.length, text = ""; i < len; i++) {
  text += cars[i] + "<br>";
```

Test your new functionality by writing a test method that deliberately moves the robot off the map. If your code works, it throws an `IllegalMoveException`, so the test won't be able to check `Kiva`'s state variables. Instead, we'll have to print an error message if the exception doesn't happen. An example is provided below:

```
    public void testMoveOutOfBounds() {
        Kiva kiva = new Kiva(defaultMap);
        kiva.move(KivaCommand.FORWARD);
        kiva.move(KivaCommand.TURN_LEFT);
        kiva.move(KivaCommand.FORWARD);
        kiva.move(KivaCommand.FORWARD);
        System.out.println("testMoveOutOfBounds: (expect an IllegalMoveException)");
        kiva.move(KivaCommand.FORWARD);

        // This only runs if no exception was thrown
        System.out.println("testMoveOutOfBounds FAIL!");
        System.out.println("Moved outside the FloorMap!");
    }
```

Implement and test obstacle checking in the forward moving helper method. If the new location passes the on-floor check, call `getObjectAtLocation()` to determine what the robot would be moving into. If it's an `OBSTACLE`, throw an `IllegalMoveException`. Create a test that moves into an obstacle and prints a failure message if no exception is thrown.

Implement and test pod collision checking in the forward moving helper method. If `Kiva` is carrying a pod, and the new location is also a `POD`, throw an `IllegalMoveException`. Create a test that moves into a pod while carrying a pod and prints a failure message if no exception is thrown.

Implement and test checking for presence of a pod in the `TAKE` helper method. If the `TAKE` command is issued while the current location is not a `POD`, throw a `NoPodException`. Create a test that tries to take a pod on an empty space and prints a failure message if no exception is thrown.

Food For Thought: It would also be impossible to pick up a pod if the robot were already carrying a pod. Is there any way to reach this condition? Should we test for it?

Implement and test existence checking in the pod drop helper method. If the current location is not a drop zone, throw an `IllegalDropZoneException`. Create a test that picks up a pod, drops it in an empty space, and prints a failure message if no exception is thrown.

Implement and test futility checking in the pod drop helper method. If a `DROP` command is issued while the current location is a drop zone, but the robot is not carrying a pod, throw an `IllegalMoveException`. Create a test that moves to a drop zone without a pod, drops on the drop zone, and prints a failure message if no exception is thrown.

Food For Thought: Does it make any difference which order we implement the checks in the pod drop helper? Does having checks with different results mean our method now has to make decisions on multiple conditions? What should we test? Is `NoPodException` a better fit when attempting to `DROP` while not carrying a pod?

5. Make More Informative Error Messages

Things are working great, but we are getting feedback from our users that it's hard to tell what is going on when things fail. We already include a message in each exception; we need to include more data in the message. Examine each of your exception messages and update them with more information where appropriate.

Let's consider the `NoPodException`. Our helper method for taking the pod might check whether it's on the appropriate space like this:

```
        FloorMapObject terrain = map.getObjectAtLocation(currentLocation);
         if (terrain != FloorMapObject.POD) {
             throw new NoPodException("No pod here!");
         }
```

We could improve this experience for our users with a better description, including the command we were executing and the location of the robot.

```
         if (terrain != FloorMapObject.POD) {
             throw new NoPodException("No pod to TAKE from location " +
 currentLocation + "!"));
         }
```

While this will work, `String.format()` is more readable, more powerful, more efficient, and easier to modify if we want to make changes later. Read only the (short) **Replace a Placeholder in a String** section from *How to Format a String, Clarified!* for the basic details you need.

The only wrinkle to keep in mind is that objects will be automatically converted to `String`s (via their `toString()` method) to fit the %s format. With this in mind, here's a better exception message:

```
         if (terrain != FloorMapObject.POD) {
             throw new NoPodException(String.format(
                 "Can't TAKE: location %s is %s, not POD!", currentLocation,
 terrain));
         }
```

That will print something like 'Can't TAKE: location (3, 1) is EMPTY, not POD!' because the `Point` and `FacingDirection` objects get converted to reasonable strings. The user now knows which command went wrong, where the robot was when it happened, and how the condition differed from what was expected.

Implement better messages for all the exceptions you throw. Since you already have tests that throw the exceptions, **re-run the tests** to verify that your `String.format()` calls work the way you expected.

6. Add MotorLifetime to the Kiva Class

The motor on each Kiva robot is rated for 20,000 hours. After this point the robot needs to have its motor replaced or be retired. We want to track how many hours a Kiva robot's motor has been running and use a field called `motorLifetime` to store the motor lifetime in milliseconds. When a Kiva robot moves forward, turns left, or turns right, the motor lifetime increments by 1000 milliseconds (1 second).

We want to track this information in milliseconds for diagnostic purposes. There are more primitive data types in Java than the ones you learned in the Duke courses. Read more about data types here . What data type should you use to store the motor's lifetime?

Let's start by adding a field called `motorLifetime` to your `Kiva` class (You should choose an appropriate data type from the reading). Now we can add a `getMotorLifetime()` method to return the current value, and an `incrementMotorLifetime()` method to add 1000 milliseconds to the current value.

Test this by:

- creating a `KivaMotorLifetimeTester` class with one method inside it to execute the following
- instantiate a `FloorMap` with the following map:

```
-----
|K D|
| P |
|* *|
-----
```

- instantiate a `Kiva` using the floor map
- print out the result of `getMotorLifetime()`. It should start at 0.
- turn right
- print out the result of `getMotorLifetime()`, It should be 1000.
- go forward
- print out the result of `getMotorLifetime()`, It should be 2000.
- turn right
- print out the result of `getMotorLifetime()`, It should be 3000.
- go forward
- print out the result of `getMotorLifetime()`, It should be 4000.
- take
- print out the result of `getMotorLifetime()`, It should be 4000.

7. Write Documentation

Future developers are also going to be working on converting your remote control to a self-driving feature. We want to make sure we leave the code base well documented. To get up to speed, read a little about Javadoc .

Implement Javadoc comments to the classes and methods in `RemoteControl`, `Kiva` and `KivaCommand`. Be sure to at least include `@param` and `@return` tags where appropriate, but feel free to experiment with others as well! Clean out the existing Javadoc you received on `RemoteControl` and replace it with something more useful to future developers. (References to “Implement the run() method” etc. were helpful instructions to you in the state you found the KivaWorld project, but not for posterity).

8. Convert User Input to an Array of `KivaCommand`s

The `RemoteControl` class currently takes in a `String` of commands from the user that looks like "FFFTRF". However, our Kiva robots cannot understand instructions in that form. They can simply make one move at a time, accepting a single `KivaCommand`. So, we'll need to take the user inputted commands and convert them into a more Kiva friendly format, a list of `KivaCommand`s that we can send to the robot's move method one at a time. Before we do that, let's take a look at a couple Java keywords called `break` and `continue`. These may be helpful in programming the task we have just described.

Learn About `break` and `continue`

Please read this short resource on what `continue` and `break` statements are: Break and Continue Statement in Java - Begin With Java .

Can you answer these quiz questions about `continue` and `break` statements?

1. What does the following code print out?

```
String[] names = {"Jessica", "Aliyah", "Joe", "Mohammed"};
for (int i = 0; i < 4; i++) {
    String name = names[i];
    if (name.equals("Joe")) {
        continue;
    }
    System.out.print(name + " ");
}
```

A) Jessica Aliyah Joe
B) Jessica Aliyah
C) Jessica Aliyah Mohammed
D) Jessica Aliyah Joe Mohammed
E) Jessica Aliyah Mohammed, and then an exception is thrown

Answer

Hide
C
Explanation: We iterate i from 0 inclusive to 3 inclusive, and when i is 2 the name is “Joe” so we skip over it. Hence we print out the first two names, skip the third, print out the fourth, and end.

2. What does the following code print out?

```
String[] names = {"Jessica", "Aliyah", "Joe", "Mohammed"};
for (int i = 0; i < 4; i++) {
    String name = names[i];
    if (name.equals("Joe")) {
        break;
    }
    System.out.print(name + " ");
}
```

A) Jessica Aliyah Joe
B) Jessica Aliyah
C) Jessica Aliyah Mohammed
D) Jessica Aliyah Joe Mohammed
E) Jessica Aliyah Mohammed, and then an exception is thrown

Answer

Hide

B
Explanation: We iterate i from 0 inclusive to 3 inclusive, and when i is 2 the name is “Joe” we break and stop the loop. Hence we only print out the first two names.

Write the Convert Method Using `continue` and/or `break`

Create a helper method in the `RemoteControl` class called `convertToKivaCommands()`. This method should take a `String` as a parameter, which will be the commands that the user types into the console (for example “FFFTRF”). It should return an array of `KivaCommand`s (in this case `FORWARD`, `FORWARD`, `FORWARD`, `TAKE`, `TURN_RIGHT`, `FORWARD`). We recommend doing this by using the `values()` method of the `KivaCommand` enum to get an array of all of the `KivaCommand`s. You can then use the `getDirectionKey()` method to determine which `KivaCommand` each char in the `String` should be converted to. If the user enters a character that does not correspond to a command throw an `IllegalArgumentException` with a useful error message.

Note: When you want to print out the contents of a `KivaCommand[]` array variable called commands, for example, you would call `Arrays.toString(commands)` to turn the array into a single `String`. You'll need to import `java.util.Arrays` to use this method. Here is some sample code that you can play with in BlueJ to demonstrate this, then use the approach for your needs:

```
import java.util.Arrays;

public class PrintArray {
    public void testPrinting() {
        String[] array = {"Cat", "Dog", "Shark"};
         // Here is a failed attempt to print out the array. It looks garbled and
not useful!
         System.out.println(array);
         // Here is a successful attempt to print out the array.
         System.out.println(Arrays.toString(array));
    }
}
```

Test Cases

To test your new method create a `RemoteControlTest` class where you will write test methods for the following cases.

Input: “FFFTRF”
Output: {FORWARD, FORWARD, FORWARD, TAKE, TURN_RIGHT, FORWARD}

Input: “B”
Output: java.lang.IllegalArgumentException: Character 'B' does not correspond to a command!

9. Fix the `NullPointerException` and Finish `RemoteControl`

Right now, when we attempt to call the `run()` method in `RemoteControl`, it doesn't seem to work. What exception is being thrown and where is it thrown from? Let's start with fixing that. If you need a little help getting started, this documentation might be helpful.

Now, we are going to fill in the rest of the `run()` method in the `RemoteControl` class. We will create a `Kiva`, using the `FloorMap` created from the file, then print some diagnostic information to the user telling them the start location of the `Kiva` and what direction it is pointing. We will then convert the user input that is collected by the `KeyboardResource` into `KivaCommand`'s using the `convertToKivaCommands()` method you wrote previously. Finally, we will have the Kiva robot `move()` each of these `KivaCommands`, and inform the user of the result!

Once you're able to get the `Kiva` to move around, at the end of a sequence of commands you need to figure out if the `Kiva` successfully picked up the pod and dropped it in the correct place. You can use `isSuccessfullyDropped()` on the `Kiva` to determine if you dropped the pod in the right place. Moreover, if you get any exceptions you know the Kiva robot failed for some reason. Your improved and informative exception messages from project task five will provide users with enough information to know what went wrong.

- If the Kiva was successful, print “Successfully picked up the pod and dropped it off. Thank you!”
- If the Kiva was not successful because it did not pick up the pod or moved after dropping it off, print “I'm sorry. The Kiva Robot did not pick up the pod and then drop it off in the right place.”. The last action the Kiva takes must be to drop the pod in the right location. It cannot move away from the drop zone.
- If the Kiva was not successful because an exception was thrown, you will see the exception come out in the Terminal with your informative exception message.

Test Cases

Execute the `run()` method with the following inputs. Validate that you see the expected output.

Inputs: Floor map file: sample_floor_map1.txt, Command string: “RFFFFFTFFFFFFFD”
Output: “Successfully picked up the pod and dropped it off. Thank you!”

Inputs: Floor map file: sample_floor_map2.txt, Command string: “RFFFFFFLFFFTRFFRFFFD”
Output: “Successfully picked up the pod and dropped it off. Thank you!”

Inputs: Floor map file: sample_floor_map3.txt, Command string: “RRFFFLFFFFLFFFRFTFRFFFLFFFFFFFFFFLFFFD”
Output: “Successfully picked up the pod and dropped it off. Thank you!”

Inputs: Floor map file: sample_floor_map2.txt, Command string: “RFFFFFFLFFFTRFFRFFFDR”
Output: “I'm sorry. The Kiva Robot did not pick up the pod then drop it off in the right place.”
(Notice these are the same commands as the successful test case above except you turn after dropping the pod. If dropping the pod in the right place is not the last command of your Kiva then this is a test case failure).

Inputs: Floor map file: sample_floor_map3.txt, Command string: “R"
Output: “I'm sorry. The Kiva Robot did not pick up the pod then drop it off in the right place.”

Inputs: Floor map file: sample_floor_map3.txt, Command string: “RRFFFLFFFFLFFFRFT"
Output: “I'm sorry. The Kiva Robot did not pick up the pod then drop it off in the right place.”

Inputs: Floor map file: sample_floor_map2.txt, Command string: “RFFFFFLFFFTRFFRFFFD”
Output: “NoPodException: Can't take nonexistent pod from location (8,1)!”

Inputs: Floor map file: sample_floor_map2.txt, Command string: “RFFFFFFLFFFTRFFRFFD”
Output: “IllegalDropZoneException: Can't just drop pods willy-nilly at (11,3)!”

Inputs: Floor map file: sample_floor_map1.txt, Command string: “F”
Output: “IllegalMoveException: Can't move onto an obstacle at (1,0)!”

•

•

Finishing the Project

You've completed all of the tasks above! Nice work getting your RemoteControl working. Congratulations!