Introduction to Karel the Robot (Karel1)

Objective: To learn about programming Karel the Robot and solve some simple problems.

Background:
In the 1970s, a Stanford graduate student named Rich Pattis decided that it would be easier to teach the fundamentals of programming if students could somehow learn the basic ideas in a simple environment free from the complexities that characterize most programming languages. Rich designed an introductory programming environment in which students teach a robot to solve simple problems. That robot was named Karel, after the Czech playwright Karel Capek, whose 1923 play R.U.R. (*Rossum's Universal Robots*) gave the word *robot* to the English language.

Karel is a very simple robot living in a very simple world. By giving Karel a set of commands, you can direct it to perform certain tasks within its world. The process of specifying those commands is called **programming**. Initially, Karel understands only a very small number of predefined commands, but an important part of the programming process is teaching Karel new commands that extend its capabilities.

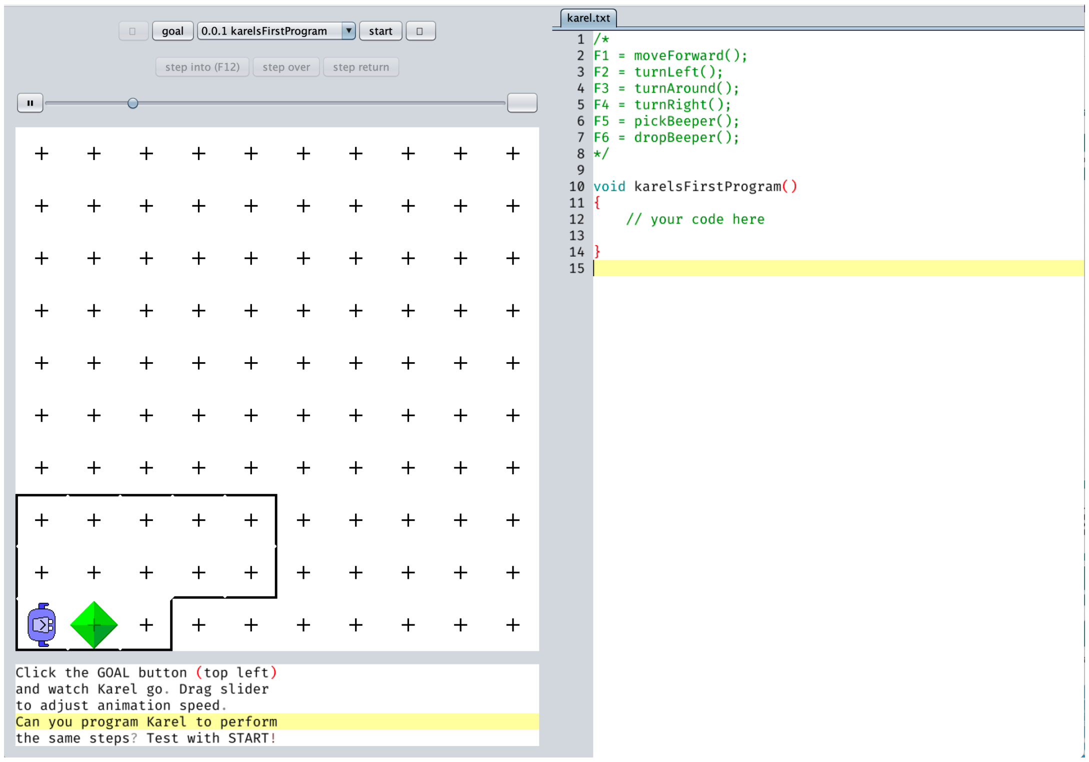
When you program Karel to perform a task, you must write out the necessary commands in a very precise way so that the robot can correctly interpret what you have told it to do. In particular, the programs you write must obey a set of **syntactic rules** that define what commands and language forms are legal. Taken together, the predefined commands and syntactic rules define the **Karel programming language**. The Karel programming language is designed to be similar to Java so as to ease the transition to the language you will be using in AP Computer Science A.

The beauty of Karel's language is its simplicity. In contrast, Java is a very sophisticated language. We can often get lost in a language's details and lose focus on the course objectives. Right now it is critical you learn problem solving techniques. Karel encourages imagination and creativity, and, hopefully, you will find Karel's exercises fun to do.

Karel's world:
Download the **karel.jar** file that contains the Karel environment. Create a directory (folder) and put the file into it. Using a PowerShell window, change to that directory and execute the following command.

```
java -jar karel.jar
```

This will bring up the following window.

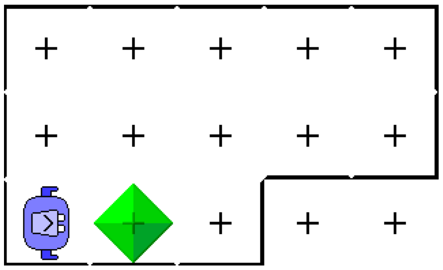


```
/*
F1 = moveForward();
F2 = turnLeft();
F3 = turnAround();
F4 = turnRight();
F5 = pickBeeper();
F6 = dropBeeper();
*/

void karelsFirstProgram()
{
    // your code here

}

```

```
Click the GOAL button (top left)
and watch Karel go. Drag slider
to adjust animation speed.
Can you program Karel to perform
the same steps? Test with START!
```

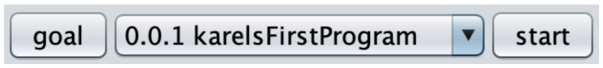
On the left side is Karel's world. Depending on the walls, the world can grow or shrink, but the maximum size is 10x10. On the right side is the programming environment. This is where you will program your solution.

Karel's world is defined by **streets** running horizontally (east-west) and **avenues** running vertically (north-south). The intersection of a street and an avenue is called a **corner**. Karel can only be positioned on corners and must be facing one of the four standard compass directions (north, south, east, west). The default Karel world is **`karelsFirstProgram`** (0.0.1) shown below. Here Karel is located at the corner of 1st Street and 1st Avenue, facing east.



Several other components of Karel's world can be seen in this example. The green object in front of Karel is a **beeper**, a plastic cone which emits a quiet beeping noise. Karel can detect a beeper if both are on the same corner or if the beeper is in front of Karel. The solid lines in the figure are **walls**. Walls serve as barriers within Karel's world. Karel cannot walk through walls and must instead go around them. Karel's world is always bounded by walls along the edges, but the world may have different dimensions depending on the specific problem Karel needs to solve.

The following controls are at the top of the Karel screen.



The center pulldown menu shows the default problem **`karelsFirstProgram`**, but it can be set to other problems. The **goal** button commands Karel to solve the current problem and gives you an idea of how it should be accomplished. The **start** button runs your code in the right-hand window and displays the result in the left-hand window. Errors will appear in the right-hand window. Your assignments will be to write code in the right-hand window that has Karel perform and complete whatever problem you are given.

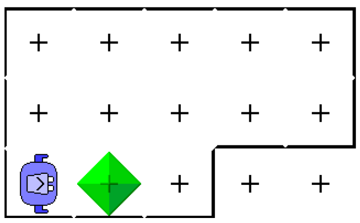
What can Karel do?
When Karel shipped from the factory, it responds to a very small set of commands:

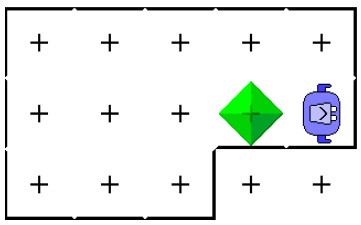
`moveForward()`	Asks Karel to move forward one block. Karel cannot respond to a **`moveForward()`** command if there is a wall blocking its way.
`turnLeft()`	Asks Karel to rotate 90 degrees to the left (counterclockwise).
`turnAround()`	Asks Karel to rotate 180 degrees.
`turnRight()`	Asks Karel to rotate 90 degrees to the right (clockwise).
`pickBeeper()`	Asks Karel to pick up one beeper from a corner and stores the beeper in its beeper bag, which can hold an infinite number of beepers. Karel cannot respond to a **`pickBeeper()`** command unless there is a beeper on the current corner.
`dropBeeper()`	Asks Karel to take a beeper from its beeper bag and put it down on the current corner. Karel cannot drop more than one beeper on a corner.

It is also important to recognize that several of these commands place restrictions on Karel's activities. If Karel tries to do something illegal, such as moving through a wall or picking up a nonexistent beeper, an **error condition** occurs. At this point, an error message appears in the right-hand program window and <u>does not execute any remaining commands</u>!

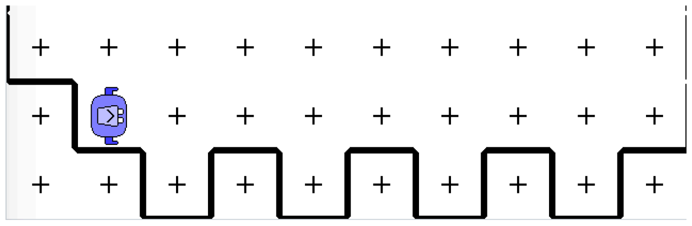
Assignment:

1. Together, we will explore the **`karelsFirstProgram`** program. We will have Karel move the beeper to a new corner while navigating around walls in the world. Karel starts on the corner of 1st Street and 1st Avenue and should end on the corner of 2nd Street and 5th Avenue. See the figures below.


Before


After

2. Next, we look at the **`fillTheHoles`** (1.2.2) problem together. Use the pulldown menu to find the problem. It should match the diagram below.



In this task, Karel fills in the holes with beepers. At first, you will do the programming on your own. After a short while, we will get back together as a class to learn about **decomposition** and create a new method called **`fillHole()`**.

3. Karel performs its commands, like **`moveForward()`**, without our help. All of Karel's internal programming is <u>hidden</u> from us and for a good reason. We do not need to know how it is programmed, just trust it works as advertised and know it will be true to our programming. This will open our discussion to **encapsulation**.