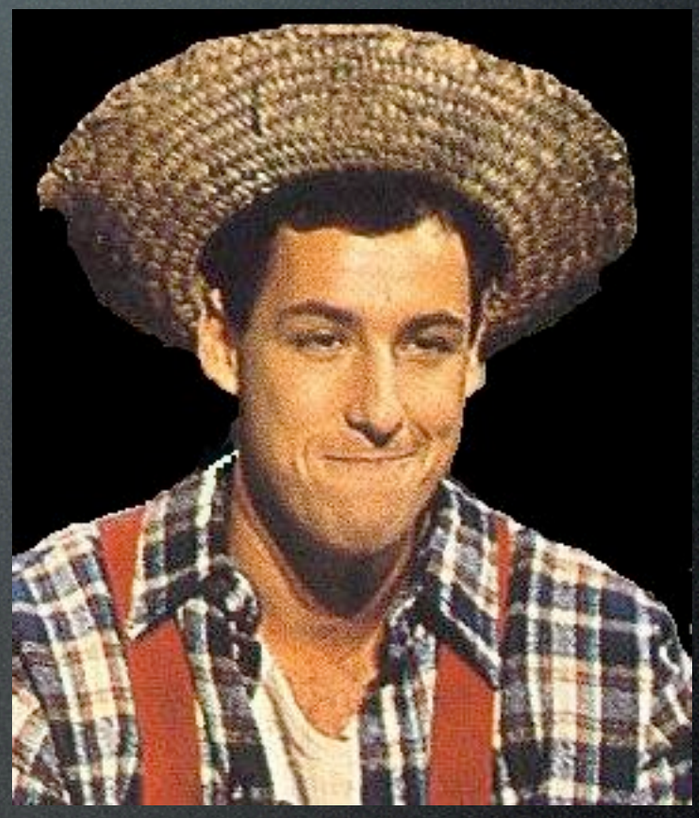
Intro to Programming For the Web

Session 2

Decomposition
Abstraction
Augmentation
Iteration
Conditions



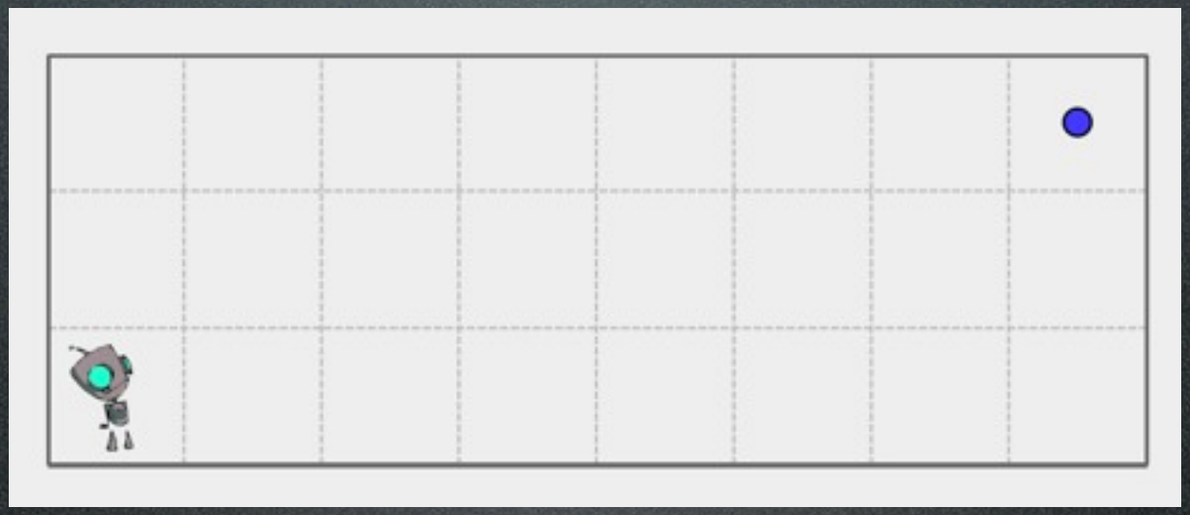
overview

- more Karel commands + exercises
- iteration, conditions, functions in Javascript
- can look at PHP if we get through those

decomposition

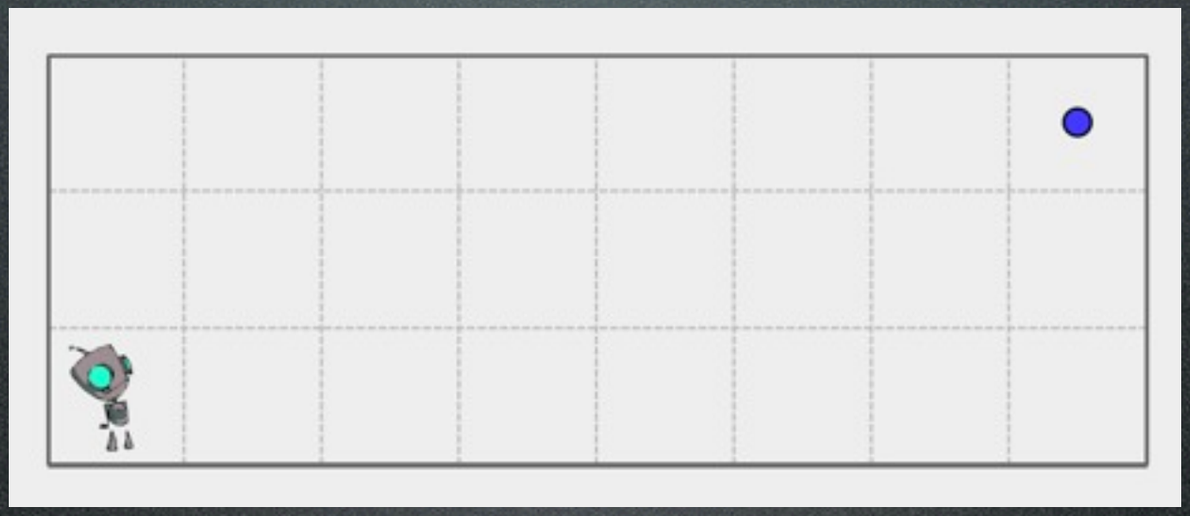
- “divide and conquer”
- take larger problem, divide it into simpler tasks

decomposition



Goal: move karel to the blue dot.

decomposition



Step 1: Move to the end of the row.
Step 2: Turn left.
Step 3: Move to the top of the column.

pseudo-code

```
// 1. move to the end of the row
// 2. turn left
// 3. move to the end of the column
```

("//" turns these lines into comments)

pseudo-code + code

```
// 1. move to the end of the row
move();
move();
move();
move();
move();
move();
move();
move();
// 2. turn left
turnLeft();
// 3. move to the end of the column
move();
move();
```

the function

- a “sub-recipe”, like chicken stock
- standard part of procedural programming

js function syntax

first you define a function:

```
function myFunctionName() {

    // function body

}
```

later, you can invoke this function:

```
myFunctionName();
```

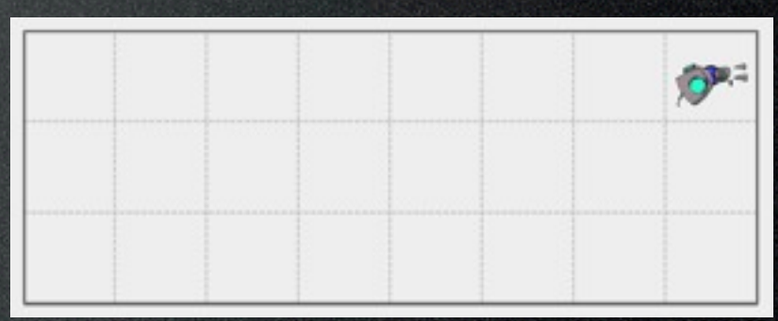
pseudo-code + code

```
// define functions

function moveToEndOfRow() {
   move();
   move();
   move();
   move();
   move();
   move();
   move();
}
// 2. turn left
turnLeft();
// 3. move to the end of the column
function moveToEndOfColumn() {
   move();
   move();
}
```

```
// solve problem

// 1. move to the end of the row
moveToEndOfRow();
// 2. turn left
turnLeft();
// 3. move to the end of the column
moveToEndOfColumn();
```



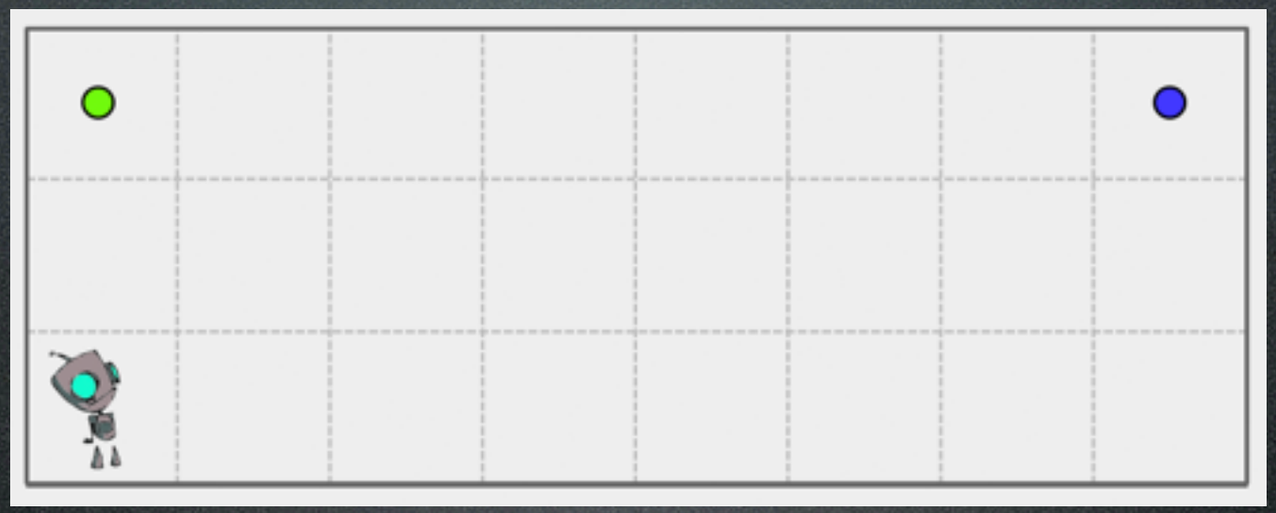
worth it?

- what did we really gain?

decomposition + abstraction

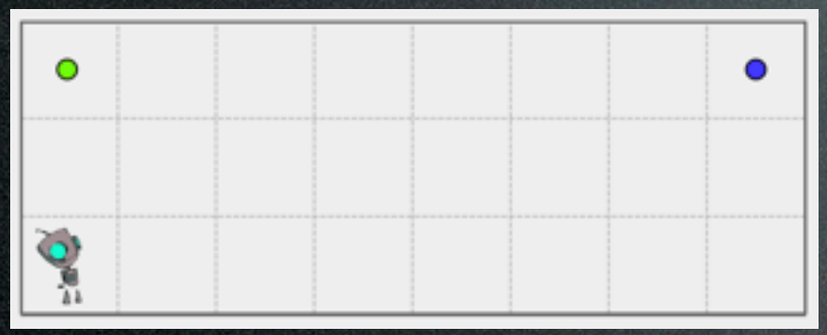
- if functions are general-purpose, they can be re-used

decomposition + abstraction



Goal: move karel to the blue dot, then to the green dot.

decomposition + abstraction



Goal: move karel to the blue dot, then to the green dot.

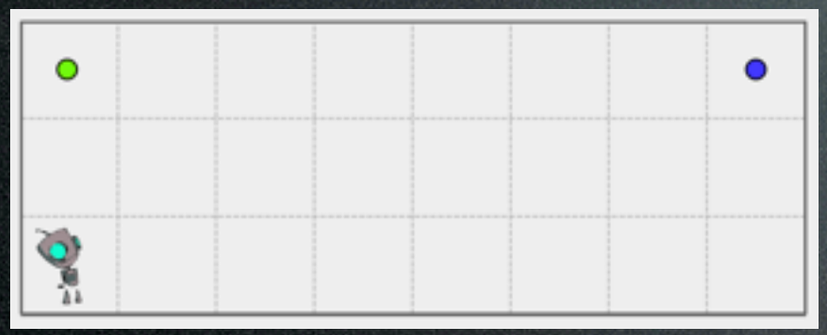
```
// pseudo-code:
// 1. move to end of row
// 2. turn left
// 3. move to top of column
// 4. turn left
// 5. move to end of row
```

decomposition + abstraction

Goal: move karel to the blue dot, then to the green dot.

```
// pseudo-code:
// 1. move to end of row
// 2. turn left
// 3. move to top of column
// 4. turn left
// 5. move to end of row
```

decomposition + abstraction

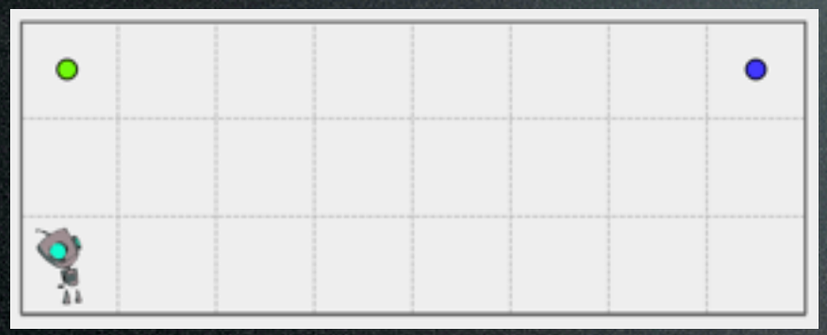


Goal: move karel to the blue dot, then to the green dot.

```
// pseudo-code:
// 1. move to end of row
// 2. turn left
// 3. move to top of column
// 4. turn left
// 5. move to end of row
```

```
// our tookkit:
moveToEndOfRow();
turnLeft();
moveToEndOfColumn();
```

decomposition + abstraction



Goal: move karel to the blue dot, then to the green dot.

```
// solution
moveToEndOfRow();
turnLeft();
moveToEndOfColumn();
turnLeft();
moveToEndOfRow();
```

augmentation

- adding your own functionality to an existing library/object

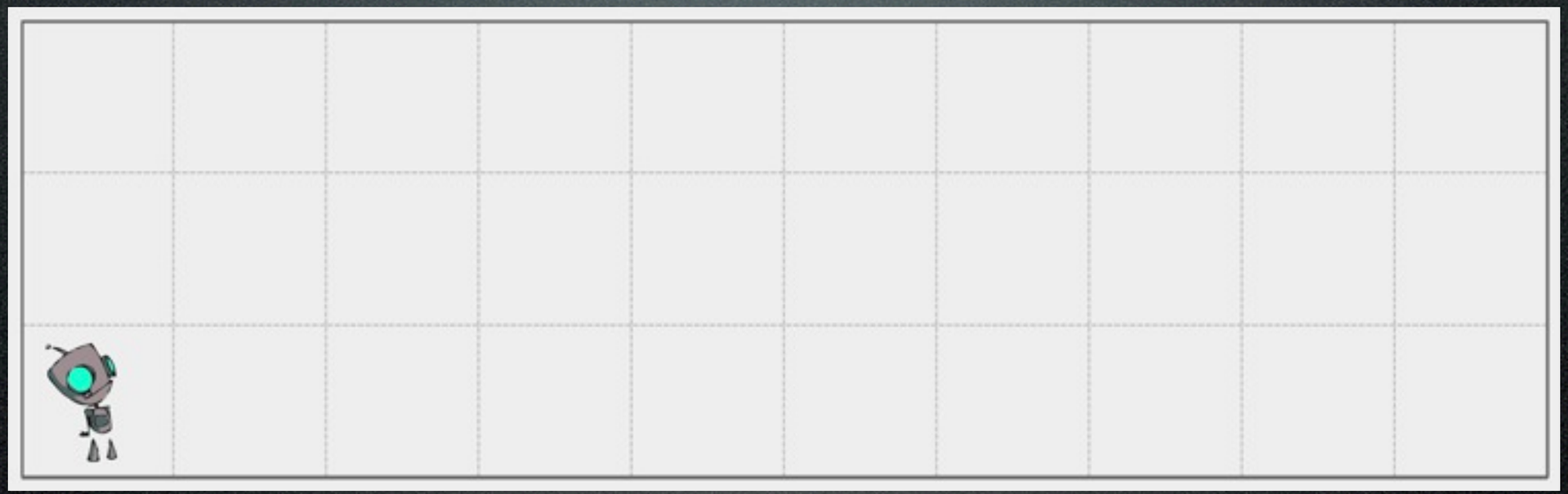
augmentation

```
function turnRight() {
  turnLeft();
  turnLeft();
  turnLeft();
}
```

download

http://ipw.patternleaf.com/karel-standalone-v1.zip

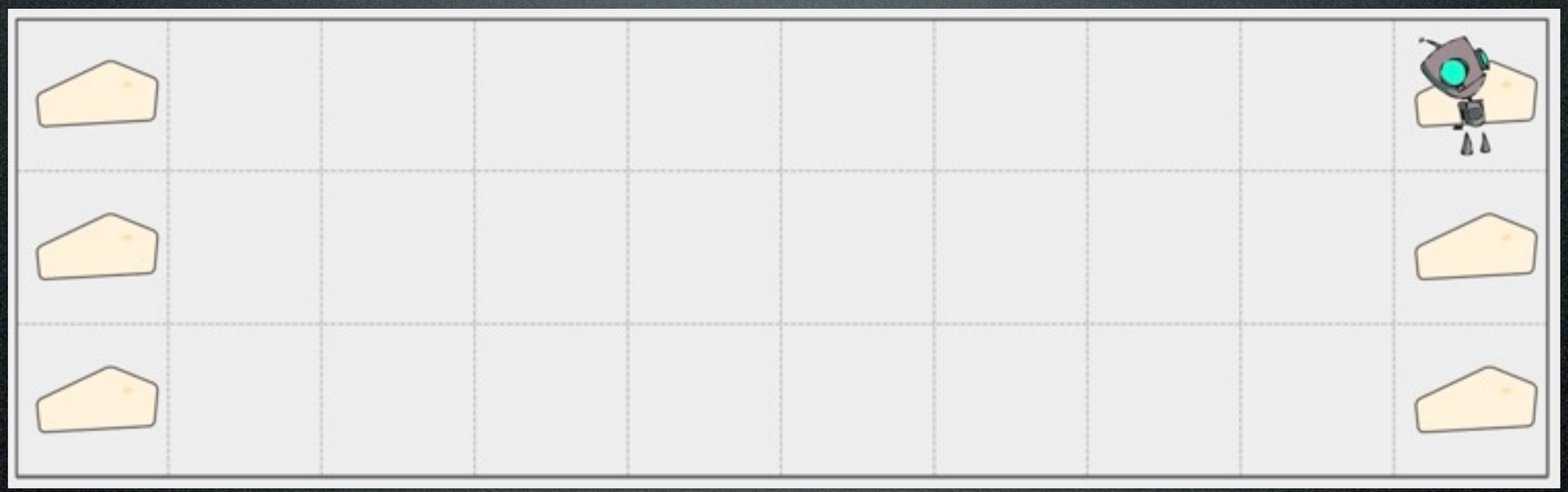
exercise



Goal: put cheese in the ends of every row.

exercise

Goal state:



Goal: put cheese in the ends of every row.

exercise

- decompose the problem
- use augmentation to define a function or two that makes the work easier

karel standalone

- install on your local machine
  - copy karel-solution-template.js
  - put solution into function passed to KarelApp.run

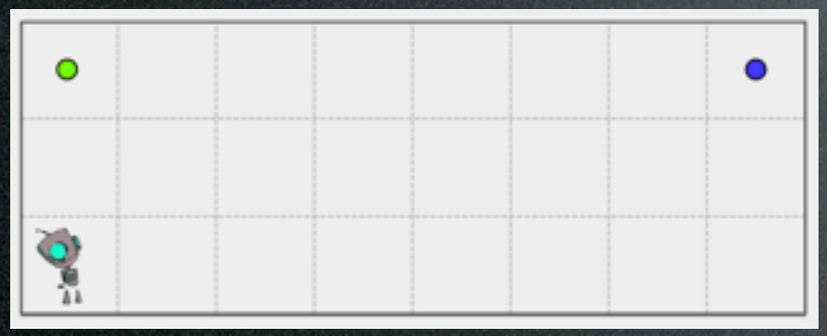
karel standalone

- uses Javascript object features

	before	using JS objects
function invocation	`move();`	`karel.move();`

augmentation	`function turnRight() {`
	`  // ...`
	`}`
		`karel.turnRight = function() {`
		`  // ...`
		`}`

our earlier solution

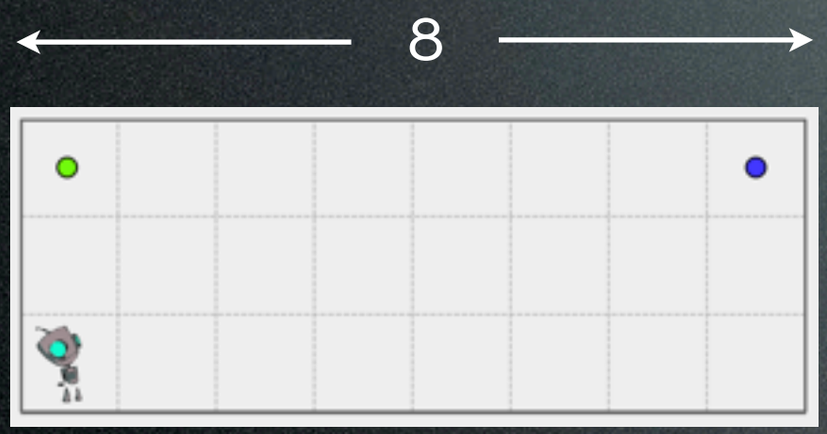


Goal: move karel to the blue dot, then to the green dot.

```
// solution
moveToEndOfRow();
turnLeft();
moveToEndOfColumn();
turnLeft();
moveToEndOfRow();
```

What's wrong with it?

our earlier solution

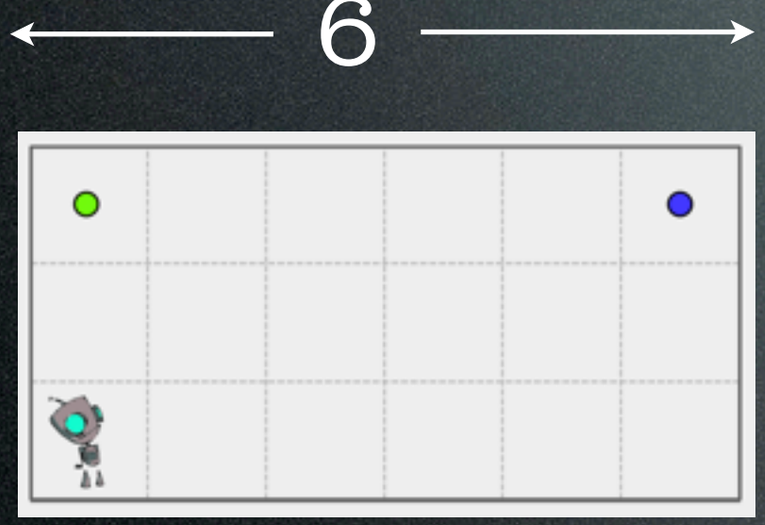


Goal: move karel to the blue dot, then to the green dot.

```
// define functions
function moveToEndOfRow() {
   move();
   move();
   move();
   move();
   move();
   move();
   move();
}
function moveToEndOfColumn() {
   move();
   move();
}

// solution
moveToEndOfRow();
turnLeft();
moveToEndOfColumn();
turnLeft();
moveToEndOfRow();
```

our earlier solution



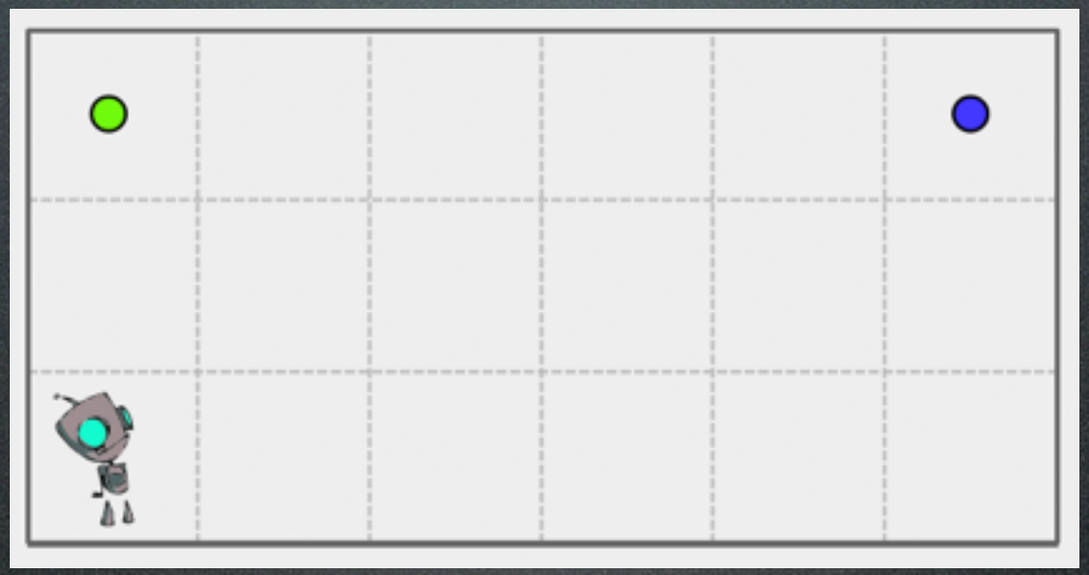
Goal: move karel to the blue dot, then to the green dot.

```
// define functions
function moveToEndOfRow() {
   move();
   move();
   move();
   move();
   move();
   move();
   move();
}

function moveToEndOfColumn() {
   move();
   move();
}

// solution
moveToEndOfRow();
turnLeft();
moveToEndOfColumn();
turnLeft();
moveToEndOfRow();
```

what's wrong with our solution?



6

abstraction

- “if functions are general-purpose, they can be re-used”

- moveToEndOfRow wasn’t very general-purpose because it only works on rows with 8 cells

- should really be called moveToEndOfRowWith8Cells

moveToEndOfRow

- more general?

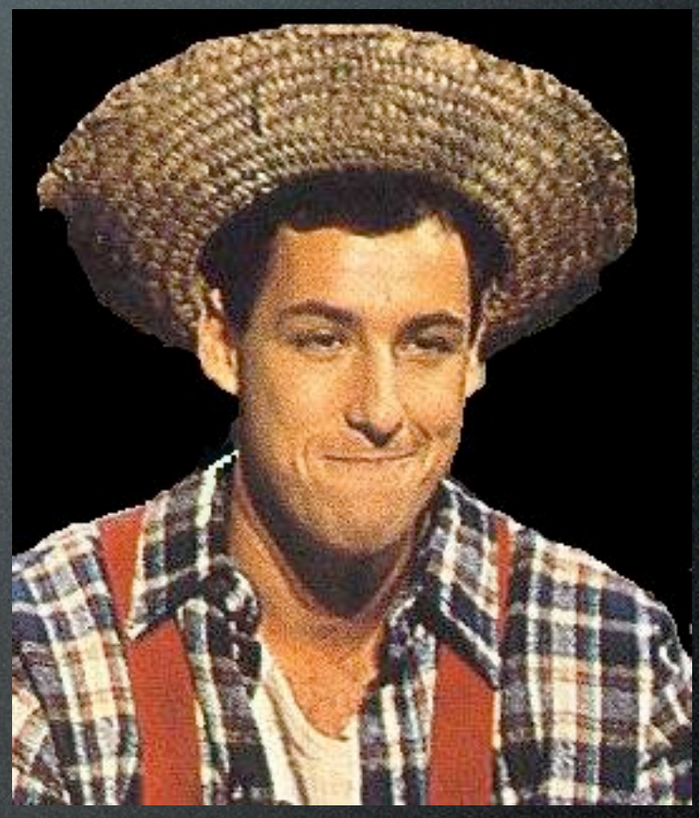
moveToEndOfRow

- “for so long as there is no wall in front of Karel, move forward”

solution:

- condition
- iteration

language features
control structures



basic conditional

```
if (<condition>) {
   <do the code in here>
}
```

basic conditional

the expression in here is evaluated as true or false.
if true, the code in the body is executed.
if false, the code in the body is skipped.

```
if (<condition>) {
   <do the code in here>
}
```

a “control structure” controls flow of code execution.

new Karel command

```
karel.frontIsClear()
```

evaluates to true or false

non-wall-colliding Karel

```
if (karel.frontIsClear()) {
   karel.move();
}
```

non-wall-colliding Karel

```
if (karel.frontIsClear()) {
   karel.move();
}
```

(only works for one move)

repeating conditional

```
while (<condition>) {
   <do the code in here forever>
}
```

repeating conditional

```
while (<condition>) {
   <do the code in here forever>
}
```

(beware infinite loops!)

repeating conditional

```
while (<condition>) {
   <do the code in here forever>
   break;
}
```

(break will break out of loop even
while condition is true)

non-wall-colliding Karel

```
while (karel.frontIsClear()) {
   karel.move();
}
```

(works anywhere!)

moveToEndOfRow

- now we can re-Christen our `moveToEndOfRowWith8Cells` with the more-manageable `moveToEndOfRow`
- or would something else be more appropriate?

moveToEndOfRow

```
karel.moveToWall = function() {
   while (karel.frontIsClear()) {
      karel.move();
   }
}
```

moveToWall

```
karel.moveToWall = function() {
    while (karel.frontIsClear()) {
        karel.move();
    }
}
```

while's brother for

```
for (<some number of times>) {
  <do the code in here>
  // optionally, break;
}
```

while's brother for

initializer ↓ end condition ↓ step ↓

```
for (var i = 0; i < 3; i++) {
  console.log('hi there ' + i);
}
```

while's brother for

initializer end condition step

```
for (var i = 0; i < 3; i++) {
  console.log('hi there ' + i);
}
```

while's brother for

initializer ↓ end condition ↓ step ↓

```
for (var i = 0; i < 3; i++) {
  console.log('hi there ' + i);
}
```

exercise 2

- exercise 1, but with a randomly-sized world
- make a copy of karel-solution-1.js
- name it karel-solution-2.js

boolean and if/else

- boolean type: true or false
- expressions can evaluate to either true or false
  - 1 = true
  - 0 = false
  - PHP and Javascript are pretty loose

boolean and if/else

- if (<boolean expression>) {

    <code to run if true>

  }

  else {

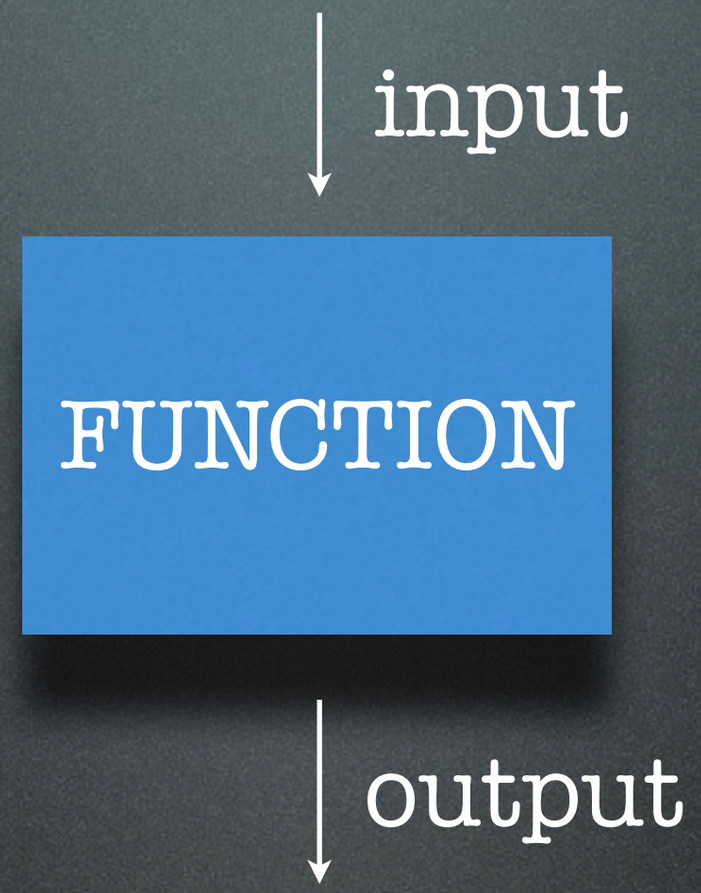
    <code to run if false>

  }

if/else alternative

- ```
  if (<boolean expression-0>) {
          <code to run if true>
  }
  else if (<boolean expression-1>) {
          <code to run if true>
  }
  ...
  else {
          <code to run if none are true>
  }
  ```
  ```

functions



function syntax

input

```
function addTwoNumbers(oneNumber, anotherNumber) {

    return oneNumber + anotherNumber;

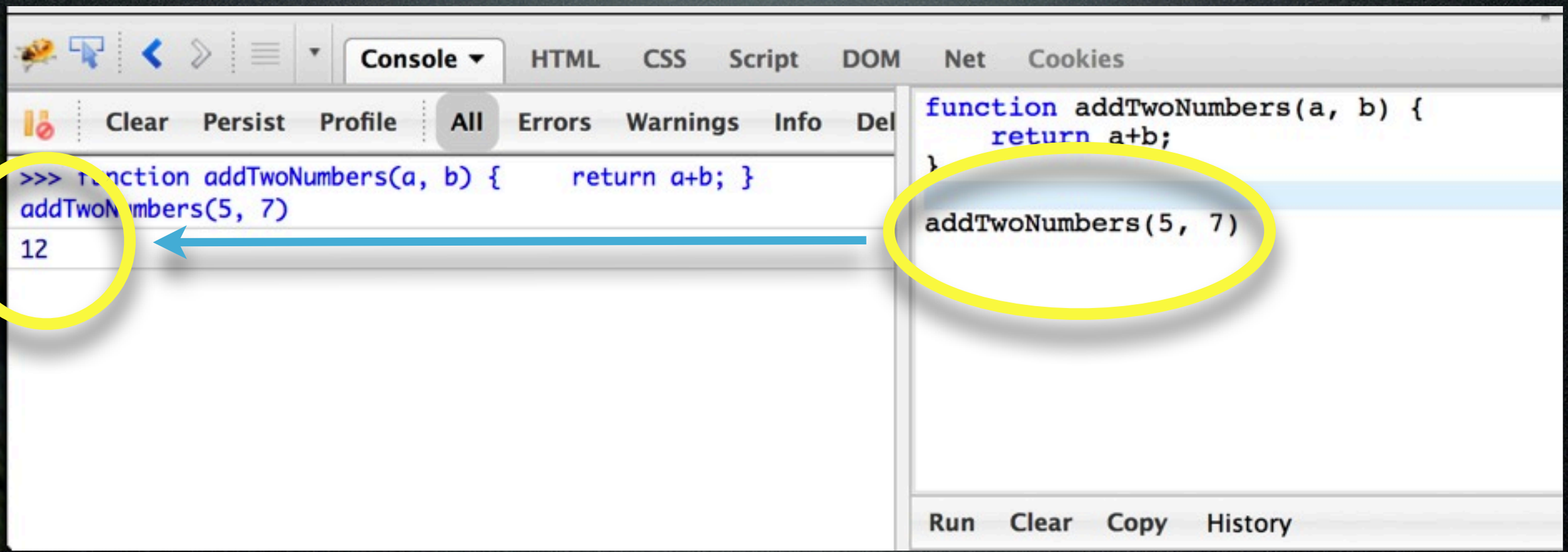
}
```

output

later:

```
addTwoNumbers(5, 7);
```

Firebug console



more new Karel commands

- cheeseIsPresent()
- isFacingNorth()
- isFacingSouth()
- isFacingEast()
- isFacingWest()

(all evaluate to true or false.
the return bools.)

Karel commands

- move
- turnLeft
- frontIsClear
- putDownCheese
- pickUpCheese
- cheeseIsPresent
- isFacingSouth
- isFacingNorth
- isFacingEast
- isFacingWest

(anything else you'll have to make yourself!)

variables quick-n-dirty

- store a result:

```
function addTwoNumbers(oneNumber, anotherNumber) {
  var result = oneNumber + anotherNumber;
  return result;
}
```

variable

isOnLedge

Write a function that returns true when Karel is to the West of a “ledge”. and facing east.

When the function ends, Karel should be in the same place he was, facing the same direction.

isOnLedge

- decompose the problem
- write pseudo-code
- replace the pseudocode with real code

isOnLedge

```
karel.isOnLedge = function() {
    // if karel is facing east
    // move forward
    // turn right
    // see if the front is clear
    // store the result of that
    // turn to face west
    // move back to where we started
    // turn around to face east again
    // return the result of what we saw
};
```

isOnLedge

```
karel.isOnLedge = function() {
    // if karel is facing east
    // move forward
    // turn right
    // see if the front is clear
    // store the result of that
    // turn to face west
    // move back to where we started
    // turn around to face east again
    // return the result of what we saw
};
```

- move
- turnLeft
- frontIsClear
- putDownCheese
- pickUpCheese
- cheeseIsPresent
- isFacingSouth
- isFacingNorth
- isFacingEast
- isFacingWest

isOnLedge

```
karel.isOnLedge = function() {
    if (karel.isFacingEast()) {
        karel.move();
        karel.turnRight();
        var result = karel.frontIsClear();
        karel.turnRight();
        karel.move();
        karel.turnAround();
        return result;
    }
    return false;
};
```

Exercise 3

- Place cheese on the left edge of the pothole.
- If no pothole is encountered, move to the eastern-most wall and stop.

- move
- turnLeft
- frontIsClear
- putDownCheese
- pickUpCheese
- cheeseIsPresent
- isFacingSouth
- isFacingNorth
- isFacingEast
- isFacingWest

Augmentation

```
karel.add = function(a, b) {
  var result = a + b;
  return result;
}
```

Iteration

```
while (<condition>) {
    // do something
    // optionally, break;
}
```

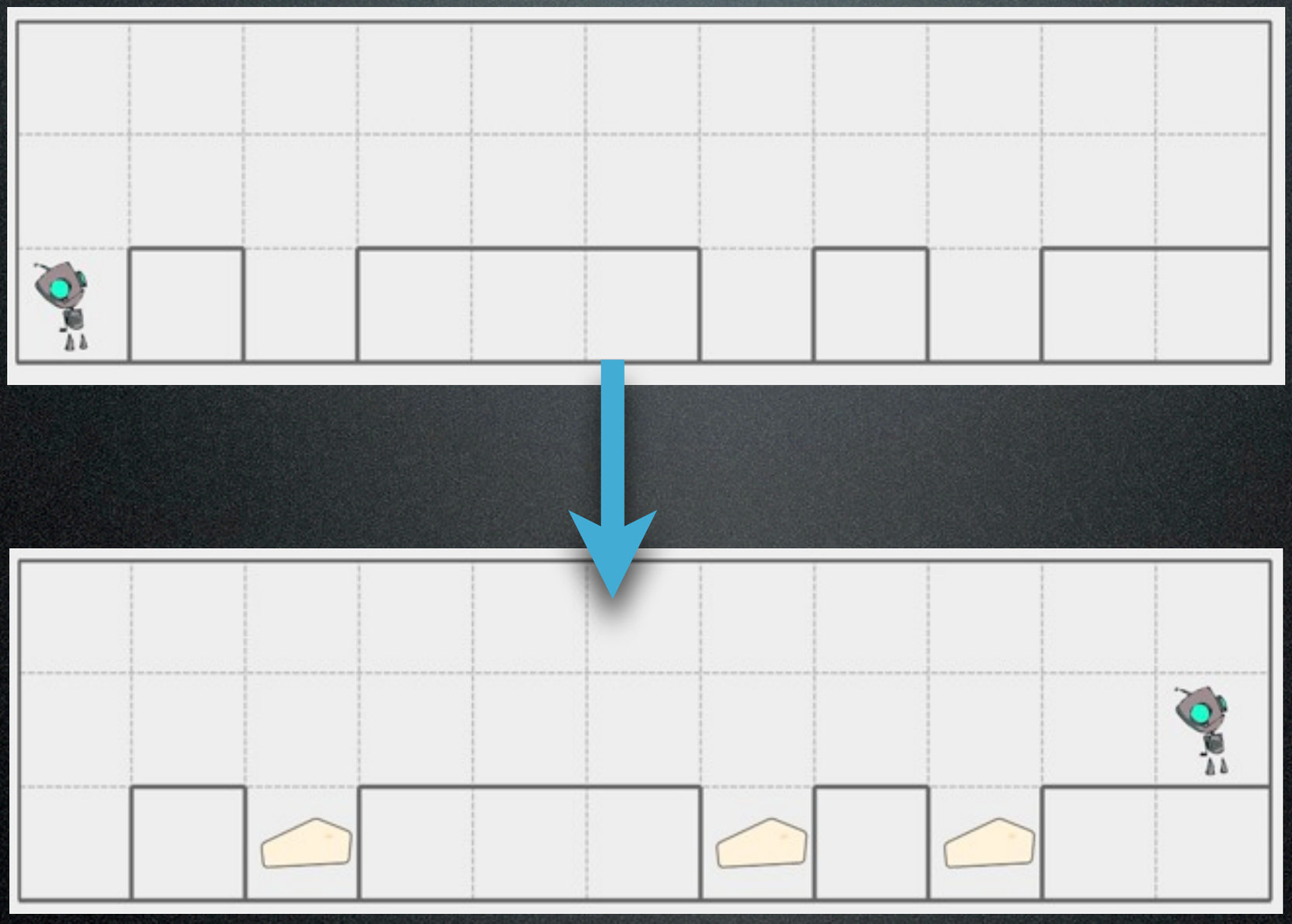
Condition

```
if (<condition>) {
    // do something
}
else {
    // do something else
}
```

(anything else you'll have to make yourself!)

Exercise 4

- Fill in every pothole with cheese.



decomposition
abstraction

- “hiding the details”
- write logic at a high level, work out details later
- “divide and conquer”

augmentation

- adding features/abilities using “atomic” or less complex features/abilities

iteration

- doing something multiple times
- while loop
- for loop (not covered yet)

conditional execution

- if ... then ... else

Decomposition
Abstraction
Augmentation
Iteration
Conditions