

Seven Steps for Solving Programming Problems

Video 1. Solving Programming: A Seven Step Approach

Solving Programming Problems

Problem Statement

I want to be able to take a foreground image and a background image and...

Big Leap: Break It Down

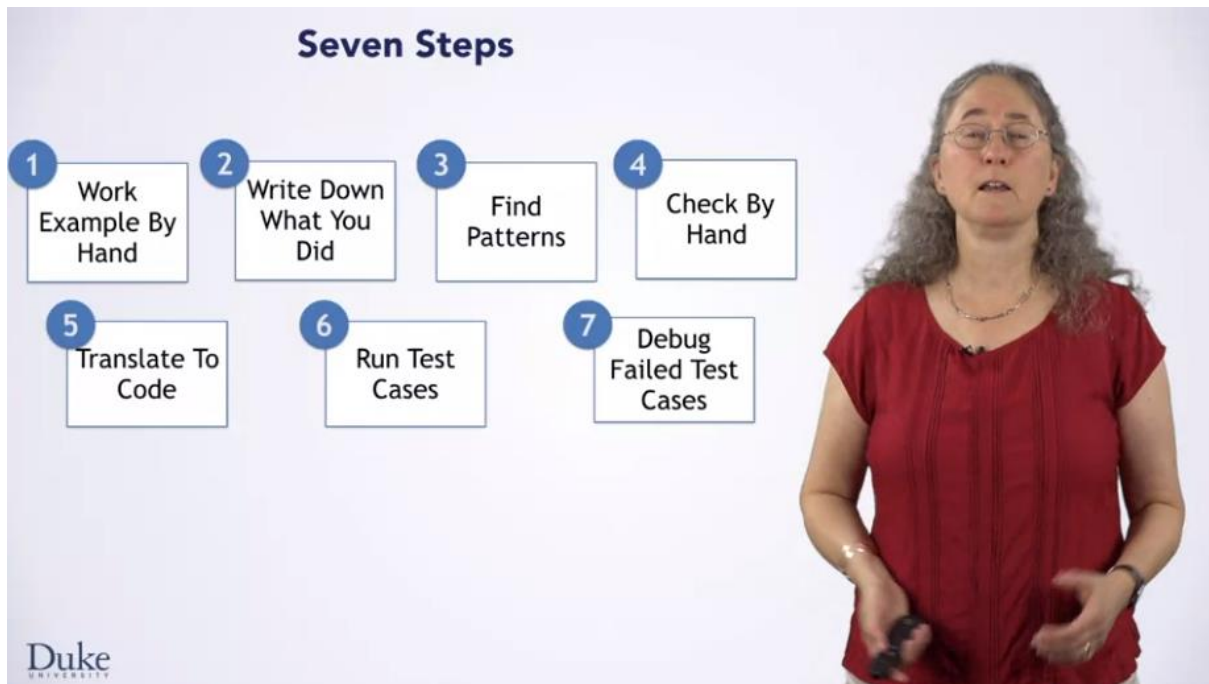
Working Code

```
var fgImage = new SimpleImage("drewRobert.png");
var bgImage = new SimpleImage("dinos.png");
var output = new SimpleImage(fgImage.getWidth(), fgImage.getHeight());

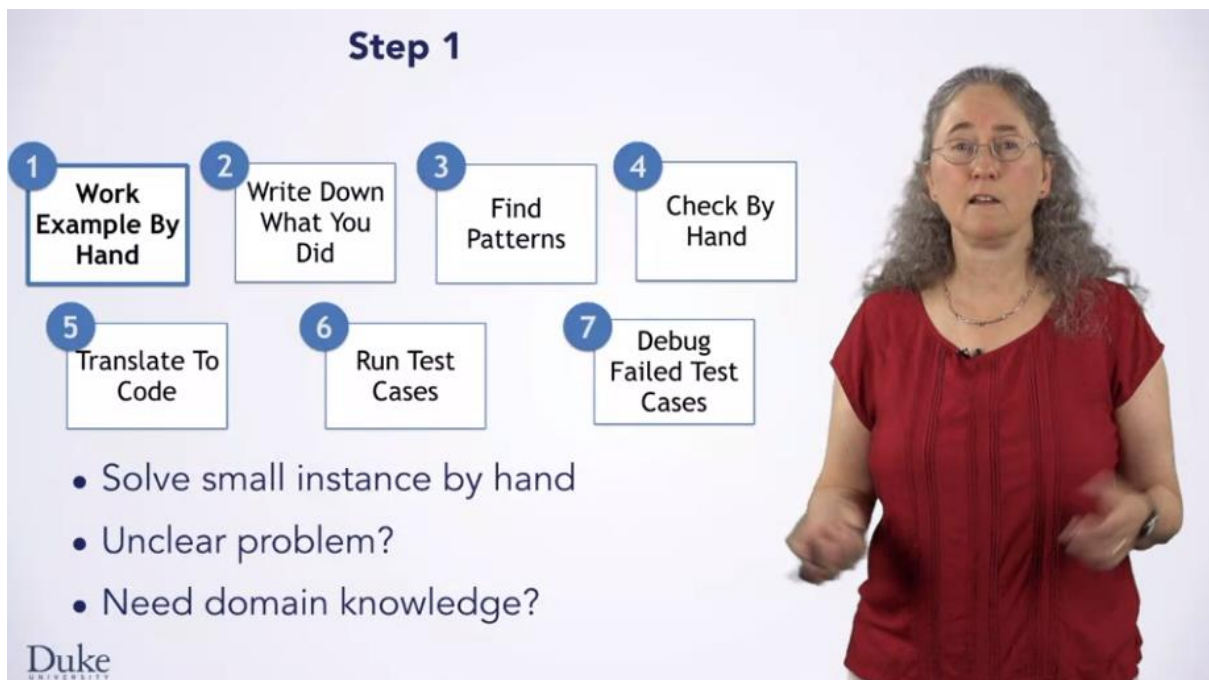
for (var pixel of image.toArray()) {
    x = pixel.getX();
    y = pixel.getY();
    r = pixel.getRed();
    g = pixel.getGreen();
    b = pixel.getBlue();
    if (g > r + b) {
        var pixel2 = bgImage.getPixel(x, y);
        output.setPixelAt(x, y, pixel2);
    }
    else {
        output.setPixelAt(x, y, pixel);
    }
}
print(output);
```

Duke UNIVERSITY

Today, you're going to learn how to solve programming problems using the seven-step approach that we will use throughout the rest of this course. When you're solving a problem, you're going to start with a problem statement. You know that you want to end up with working code but going straight from a problem statement to working code is a rather large leap. It can take some significant thought and work. This is why we break it down into seven steps for you, giving you a manageable approach where you know what to do next to solve the problem.



This is a seven-step process that we recommend you use whenever you are solving programming problems that are difficult. As your programming skills improve, many problems will become easy enough to just do them in your head. However, there will always be some problems that are harder, and having a step-by-step approach will be helpful to you. Let us look at each step in a bit more detail.



The first step is to work an example of the problem yourself, by hand. This should be a small instance of the problem, something with about four or five pieces of data to manipulate. You don't want to try to process a million pieces of data yourself, that would take forever. If you have trouble doing the problem yourself, you cannot write a program to do it. So what should you do if you get stuck here? Well, one

of two things could be wrong. Maybe the problem is just unclear. The problem statement does not give you enough information about what you're supposed to do. In a classroom setting, you could consult your teacher or TA. In a professional setting, you might work with your technical lead or customer to clarify the requirements, or you might just need to refine the problem statement yourself. The other potential problem is that you lack domain knowledge, the knowledge of the field that the problem belongs to. If you're trying to write a program to compute physical motion and you do not know the physics equations that you need, that would be a lack of domain knowledge. When you have this sort of problem, you need to find the relevant domain knowledge before you proceed.

Step 2

- 1 Work Example By Hand
- 2 **Write Down What You Did**
- 3 Find Patterns
- 4 Check By Hand
- 5 Translate To Code
- 6 Run Test Cases
- 7 Debug Failed Test Cases

- Write down exact steps
- Just that instance
- Tricky: Do without thinking

Duke UNIVERSITY

In step two, you want to write down what you just did to solve this problem. You want to be as exact as possible. Do not leave anything out, and write down how you solved it in a step-by-step fashion. At this point in the process, you are just writing down the step-by-step approach for the one particular instance you solved, not the more general problem. The tricky part in this step is that we often do things without thinking about them. If you gloss over something or you're not precise about what you did, it will make the later steps harder.

Step 3

- 1 Work Example By Hand
- 2 Write Down What You Did
- 3 Find Patterns
- 4 Check By Hand
- 5 Translate To Code
- 6 Run Test Cases
- 7 Debug Failed Test Cases

- Algorithm for any instance
- Find patterns
- Repetition, conditions, values


Duke UNIVERSITY

- Difficulties?
 - Try Step 1 + 2 again
 - Different inputs

Duke UNIVERSITY

In step three, you want to move from the particular instance that you solved in step one to an algorithm that works for any instance of the problem. That is, you want to devise an algorithm which can solve the problem correctly for any input that you give it. You'll do this by finding patterns in what you did and replacing specific behavior with more general behavior based on that pattern. Some important tools for finding patterns which we will delve into more deeply soon are looking for repetitive behavior, finding behavior which you do sometimes but not always, and figuring out under what conditions you do it, and figure out how specific values you use relate to the parameters you picked. So what should you do if you have trouble with this step? Go back and try steps one and two again. Use different inputs which will give you another example to work from. You can see the steps for a different instance of the problem and have more information to help you find the patterns.

Step 4



- 1 Work Example By Hand
- 2 Write Down What You Did
- 3 Find Patterns
- 4 Check By Hand
- 5 Translate To Code
- 6 Run Test Cases
- 7 Debug Failed Test Cases

- Incorrect pattern? Find now
- Check with different inputs

Duke UNIVERSITY

In step four, you want to check your algorithm before you proceed to turn it into code. If you found the patterns incorrectly or otherwise made a mistake in step three, you would like to find that out now. The way you take your algorithm is to pick at least one different input. Again, it should be a small one, and follow the steps of your algorithm for it. If your algorithm gives you the right answer, then you are ready to move on. If not, you should go back and fix it first.

- Translate algorithm to code
 - Programming language



Duke UNIVERSITY

Now that you have devised an algorithm to solve the problem, you're ready to translate that algorithm into code. This step is where the syntax of a specific programming language comes into play. You need to write down your steps in the syntax of that language.

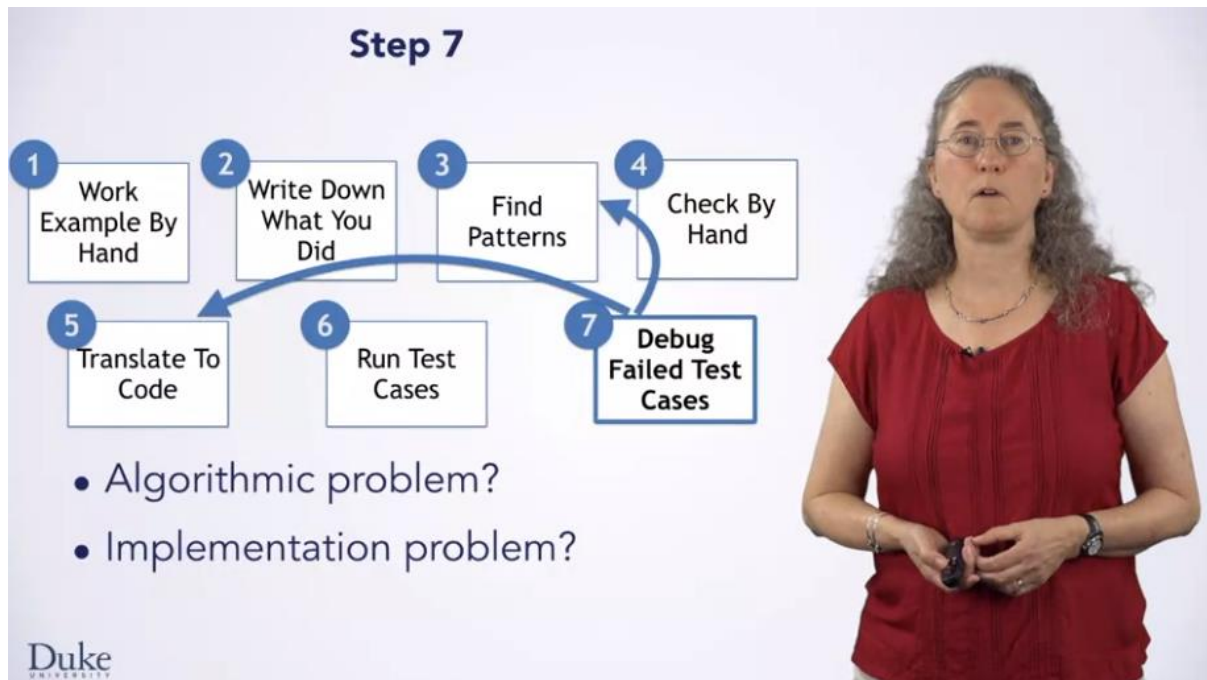
- Run test cases on code
 - Execute program
 - Check answer



Duke UNIVERSITY

Once you've written your code, you want to be sure that it works correctly, so you run test cases on it. Running a test case involves executing the code on a particular input and checking if it produced the right answer. The more test cases your code passes, the more confident you could be that it is correct.

However, no amount of testing can guarantee that the code is right. When your program fails a test case, you know something is wrong. And when that happens, it's time to debug your program. You can watch a review video about debugging if you need more information, but at a high level, you will apply the scientific method to understand what is wrong with your program and determine how to fix it.



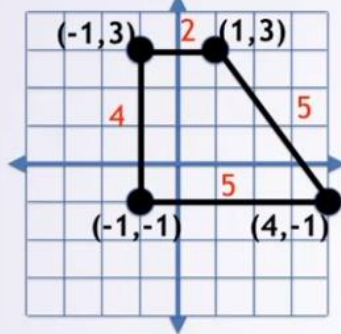
Once you have identified the problem using the scientific method, you will need to revisit a previous step to fix it. If the problem is in the algorithm you designed, you will want to go back to step three and rethink your algorithm. If your algorithm is correct but you implemented it incorrectly in code, you will want to return to step five to fix your code.

Now you have learned the seven-step approach to solving a programming problem. In the next video, we will work through an example of the process. This process can guide you through programming problems you need to solve not only throughout the rest of this course but whenever you need to solve a difficult problem. Thank you.

Video 2. Seven Steps in Action: Developing an Algorithm

Hi, in this video we're going to walk through an example of using our seven step process to solve a programming problem.

Step 1: Work an Instance Yourself



$$\begin{aligned} 4 + 5 &= 9 \\ 9 + 5 &= 14 \\ 14 + 2 &= 16 \end{aligned}$$

↑
Answer

Problem:

Given a shape, find its perimeter

Duke
UNIVERSITY



In particular, the problem we're going to work on is, given a shape, find its perimeter. Step one is to work an instance of the problem yourself. That means you'll need to draw a shape and find its perimeter. We'll need a little domain knowledge here for this problem. Specifically, what is the perimeter of a shape? If you don't recall, the perimeter is the sum of the lengths of all the sides of the shape. We'll need to know that a shape is defined by its points, and the points are listed in order as they appear around the perimeter. First, we'll draw a coordinate grid, so we can draw our shape precisely and carefully. Then we'll draw a shape on that grid. We're noting the coordinates of each point in the shape as we draw it. This will allow us to easily do math on them to compute the lengths of the edges of our shape. Once we have the shape, we can start finding its perimeter. This left edge has length 4, and this bottom edge has length 5. So we can add them together and get 9, the running total of our perimeter. The next edge is the diagonal, so we'll need to do a little bit of math. The difference in the x's is 3, and the difference in the y's is 4. The square root of 3 squared + 4 squared, 9 + 16, or 25, is 5. That's the length of this edge. We can then add 9 + 5, so that our running total is 14. Now we see that the last edge has length 2. We add 2 to 14 to get 16. So 16 is our answer for this particular instance of the perimeter problem.

Step 2: Write Down What You Just Did


1. Found distance from 1st point to 2nd point (it was 4)
2. Found distance from 2nd point to 3rd point (it was 5)
3. Added $4 + 5 = 9$
4. Found distance from 3rd point to 4th point (it was 5)
5. Added $9 + 5 = 14$
6. Found distance from 4th point to 1st point (it was 2)
7. Added $14 + 2 = 16$
8. 16 is my answer



Now you're ready for step two. Writing down specifically what we just did. First, we found the distance from the first point to the second point, which was 4. Then we took the second point to the third point, which was 5. Then we added $4 + 5$ to get a running total of 9. Next, we found the distance from the third point to the fourth point, which was 5. And added $9 + 5$, so our running total is 14. Then we found the distance from the fourth point back to the first point, which is 2, and we added $14 + 2$ to get 16. Last, we said that 16 was our answer. So we can write down the steps to solve this particular instance of the problem, as you can see here.

Step 3: Generalize

1. Found distance from 1st point to 2nd point (it was 4)
1a. Added $0 + 4 = 4$
2. Found distance from 2nd point to 3rd point (it was 5)
3. Added $4 + 5 = 9$
4. Found distance from 3rd point to 4th point (it was 5)
5. Added $9 + 5 = 14$
6. Found distance from 4th point to 1st point (it was 2)
7. Added $14 + 2 = 16$
8. 16 is my answer



Duke UNIVERSITY

Now we're ready to move on to step three, where we're going to find the patterns and generalize to find the perimeter of any shape, not just the one we just saw. One thing you might notice is that we're doing almost the same thing repeatedly. When we generalize, we want to look for similar steps and express them as **repetition**. To do this, we'll need to make them match up exactly. Which we might do here by starting out by adding $0 + 4$ to get 4. Why does this seem like a good way to make these match up? We'll keep adding the previous result to our running total of all the lengths. So it makes sense to start with 0 for our running total, and add our current result to it.

Step 3: Generalize

1. Find distance from 1st pt to 2nd pt, name it **currDist**
- 1a. Add $0 + \text{currDist} = 4$
2. Find distance from 2nd pt to 3rd pt, name it **currDist**
3. Add $4 + \text{currDist} = 9$
4. Find distance from 3rd pt to 4th pt, name it **currDist**
5. Add $9 + \text{currDist} = 14$
6. Find distance from 4th pt to 1st pt, name it **currDist**
7. Add $14 + \text{currDist} = 16$
8. 16 is my answer

Duke
UNIVERSITY



The next thing we might do in generalizing this algorithm is give this quantity a name. It won't always have these values. So we should name the quantity and refer to it by that name. We'll call it **currDist**, since it's the current distance. When we name this quantity currDist, we'll also want to change all the places we used that value when we computed it to reflect the name that we just chose.

Step 3: Generalize

1. Find distance from 1st pt to 2nd pt, name it currDist
- 1a. Add $0 + \text{currDist} = 4$
2. Find distance from 2nd pt to 3rd pt, name it currDist
3. Add $4 + \text{currDist} = 9$
4. Find distance from 3rd pt to 4th pt, name it currDist
5. Add $9 + \text{currDist} = 14$
6. Find distance from 4th pt to 1st pt, name it currDist
7. Add $14 + \text{currDist} = 16$
8. 16 is my answer

Give this result a name (**totalPerim**)
and replace uses with name

Duke
UNIVERSITY



Next, we should give this quantity a name. Here, total parameter makes sense, or totalPerim, as this quantity is the total parameter of the shape that we're calculating as we go through the steps. Again, when we name the quantity, we'll need to replace the places we used that value using the name we just gave it, totalPerim. This gives us an algorithm that looks like this.

Step 3: Generalize

1. Find distance from 1st pt to 2nd pt, name it `currDist`
- 1a. Add `0` + `currDist` = 4
2. Find distance from 2nd pt to 3rd pt, name it `currDist`
3. Update `totalPerim` to be `totalPerim + currDist`
4. Find distance from 3rd pt to 4th pt, name it `currDist`
5. Update `totalPerim` to be `totalPerim + currDist`
6. Find distance from 4th pt to 1st pt, name it `currDist`
7. Update `totalPerim` to be `totalPerim + currDist`
8. `totalPerim` is my answer

What about this one?

Duke
UNIVERSITY



Let's stop and look at this algorithm for a second. Does anything strike you as maybe just a little bit odd? What about this place that we used 0? It isn't a previous value that we computed, but why did we put this line in to begin with? We wanted all the steps to repeat exactly, but now they look different. Can we make them look the same?

Step 3: Generalize

0. Start with `totalPerim = 0`
1. Find distance from **1st** pt to 2nd pt, name it `currDist`
- 1a. Update `totalPerim` to be `totalPerim + currDist`
2. Find distance from **2nd** pt to 3rd pt, name it `currDist`
3. Update `totalPerim` to be `totalPerim + currDist`
4. Find distance from **3rd** pt to 4th pt, name it `currDist`
5. Update `totalPerim` to be `totalPerim + currDist`
6. Find distance from **4th** pt to 1st pt, name it `currDist`
7. Update `totalPerim` to be `totalPerim + currDist`
8. `totalPerim` is my answer

Duke
UNIVERSITY




Sure, we can make `totalPerim` start at 0 before we begin the repetitive steps. Then we can just update `totalPerim` in our first step. Now these steps form a repetitive pattern. They're the same, except for which particular points they're working with. The first point in each of the repetition counts to the points of the shape in the order in which they appear. That's great, we like it when we can iterate over a sequence,

because we can ultimately express this in code with a foreach loop. But this second point in each repetition is a bit more problematic. We'd have to have a way to get to the point after the current one. Now, there are ways that we could set up our shape's interface, or API, to iterate over the points and ask for the next one too. But let's do something very clever.

Step 3: Generalize

0. Start with `totalPerim = 0`
6. Find distance from 4th pt to **1st** pt, name it `currDist`
7. Update `totalPerim` to be `totalPerim + currDist`
1. Find distance from 1st pt to **2nd** pt, name it `currDist`
- 1a. Update `totalPerim` to be `totalPerim + currDist`
2. Find distance from 2nd pt to **3rd** pt, name it `currDist`
3. Update `totalPerim` to be `totalPerim + currDist`
4. Find distance from 3rd pt to **4th** pt, name it `currDist`
5. Update `totalPerim` to be `totalPerim + currDist`
8. `totalPerim` is my answer




Duke UNIVERSITY

Let's reorder the steps, so that we do the distance from the fourth point to the first point first. That is, let us write the steps in this order. First, is it okay to do this? When we want to reorder things, we have to think about it and be very careful. Here, the reordering is totally fine. It's totally fine since addition is commutative, it doesn't matter what order we do the addition in. Second, why is it useful?

Step 3: Generalize

0. Start with `totalPerim = 0`
1. Find distance from 4th pt to **1st** pt, name it `currDist`
2. Update `totalPerim` to be `totalPerim + currDist`
3. Update **`prevPt`** to be the 1st pt.
4. Find distance from **`prevPt`** to 2nd pt, name it `currDist`
5. Update `totalPerim` to be `totalPerim + currDist`
6. Update **`prevPt`** to be the 2nd pt.
7. Find distance from **`prevPt`** to 3rd pt, name it `currDist`
8. Update `totalPerim` to be `totalPerim + currDist`
9. Update **`prevPt`** to be the 3rd pt.
10. Find distance from **`prevPt`** to 4th pt, name it `currDist`
11. Update `totalPerim` to be `totalPerim + currDist`
12. Update **`prevPt`** to be the 4th pt.
13. `totalPerim` is my answer





Duke UNIVERSITY

Well, now we're going to the points in order, giving us a natural for each repetition. But the other point, which does not lend itself for each repetition, is the one we just used. That means we can simply remember the previous point in a variable and use it with the next point. We express this idea by updating our algorithm to look like this. Notice how we update `prevPt` to be the point we just finished with before moving on to the next point. Then we make use of that point in the next set of steps. But what about this point?

Step 3: Generalize

0. Start with `totalPerim = 0`
- 0a. Start with `prevPt = the 4th point`
1. Find distance from `prevPt` pt to **1st** pt, name it `currDist`
2. Update `totalPerim` to be `totalPerim + currDist`
3. Update `prevPt` to be the 1st pt.
4. Find distance from `prevPt` to 2nd pt, name it `currDist`
5. Update `totalPerim` to be `totalPerim + currDist`
6. Update `prevPt` to be the 2nd pt.
7. Find distance from `prevPt` to 3rd pt, name it `currDist`
8. Update `totalPerim` to be `totalPerim + currDist`
9. Update `prevPt` to be the 3rd pt.
10. Find distance from `prevPt` to 4th pt, name it `currDist`
11. Update `totalPerim` to be `totalPerim + currDist`
12. Update `prevPt` to be the 4th pt.
13. `totalPerim` is mv answer





We can use the same idea we saw earlier when we initialized `totalPerim` to 0. Start `prevPt` out with the value we want before we begin repeating the steps. But will it always be the fourth point? No, it just happens that we had four points here, but in general, we'll want to start `prevPt` as the last point in our shape. Okay, great. Now we have nice repetitive steps where the only difference is the point we're working on. And those go in order through the points in the shape. So we can express all these steps in the colored boxes that you see here as a repetition for each point of the steps. This gives us a nice general algorithm for finding the perimeter of any shape. Later, we'll translate this into code.

Video 3. Seven Steps in Action: Testing the Algorithm

Hi. We've developed an algorithm to find the perimeter of a shape. But are we ready to turn this algorithm into code? Well we could, but we'd like to be confident that it's right before we do that. After all, there were a lot we had to do to generalize our steps. And it's entirely possible that we made a mistake. Or perhaps we just didn't think through all the special cases. So before we turn this into code, we should test it out.

Step 4: Test Algorithm

0. Start with `totalPerim = 0`
1. Start with `prevPt = the last point`
2. For each point `currPt` in the shape,
 3. Find distance from `prevPt` to `currPt`, name it `currDist`
 4. Update `totalPerim` to be `totalPerim + currDist`
 5. Update `prevPt` to be `currPt`.
6. `totalPerim` is my answer

Duke
UNIVERSITY

To test the algorithm, we need a different instance of the problem, something other than what we used to make the algorithm. In fact, it's good if our test instance is pretty different from the one we used to make the algorithm. Here, we've shown a triangle instead of the four-sided trapezoid we used when we developed the algorithm. Before we go any further, take a second to figure out what the right answer is. What is the perimeter of this shape? When you finish, you'll want to check if the answer to our simulated algorithm is correct. And to do, that you'll need to know the right answer. Now, we'll execute the algorithm by hand for this particular input.

Step 4: Test Algorithm

<code>totalPerim</code>	10
<code>prevPt</code>	<code>(-3,4)</code>
<code>currPt</code>	<code>(-3,4)</code>
<code>currDist</code>	10

0. Start with `totalPerim = 0`
1. Start with `prevPt = the last point`
2. For each point `currPt` in the shape,
 3. Find distance from `prevPt` to `currPt`, name it `currDist`
 4. Update `totalPerim` to be `totalPerim + currDist`
 5. Update `prevPt` to be `currPt`.
6. `totalPerim` is my answer

Duke
UNIVERSITY

Step 4: Test Algorithm

totalPerim	24
prevPt	(-3,-4)
currPt	(3,-4)
currDist	6

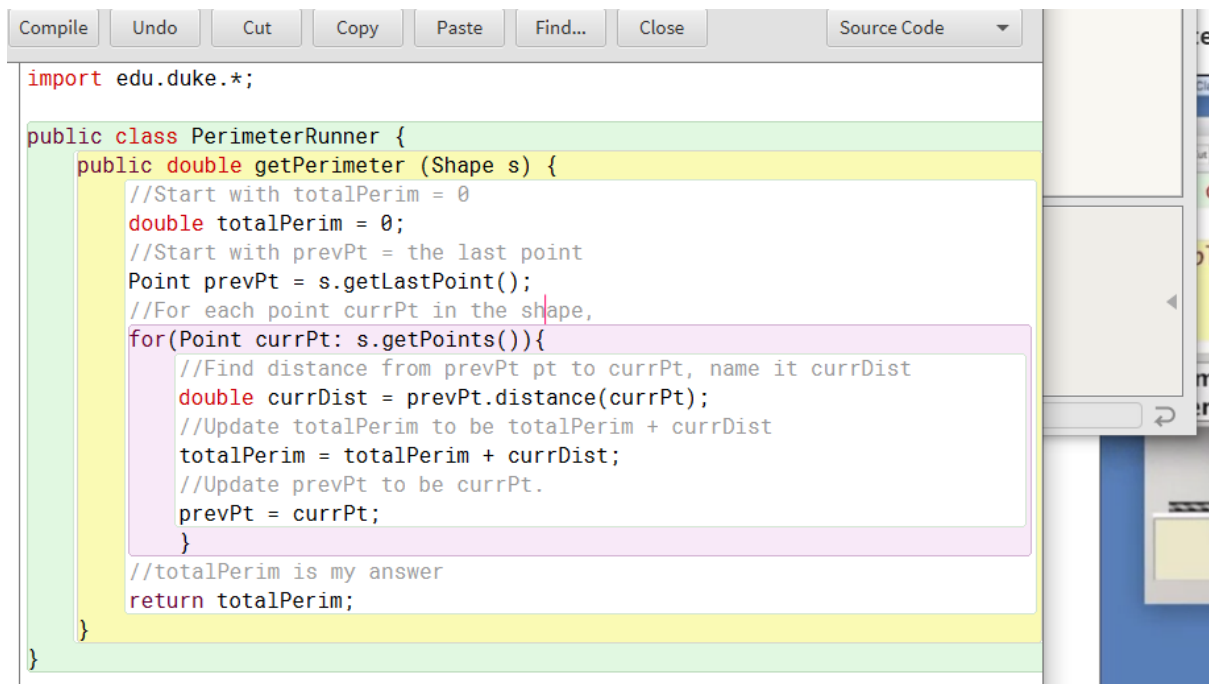
0. Start with `totalPerim = 0`
1. Start with `prevPt` = the last point
2. For each point `currPt` in the shape,
 3. Find distance from `prevPt` to `currPt`, name it `currDist`
 4. Update `totalPerim` to be `totalPerim + currDist`
 5. Update `prevPt` to be `currPt`.
6. `totalPerim` is my answer

5. Update `prevPt` to be `currPt`.

Duke
UNIVERSITY

As we test the algorithm, notice the similarities between the code and English. We're going to execute this English algorithm by hand, just as we executed code by hand. They both work pretty much the same way. And that's not a coincidence. When you turn this algorithm into code, you want to write down code that has the same semantics as the English. The same meaning. The code should transform the program state in the same way that the English transforms this diagram. So we'll start with `totalPerim` and set it to 0, and `prevPt` being the last point in the shape. But what is the last point? We'll say that points in this shape start at the top and go counter-clockwise. So this point in the lower right-hand corner is the last one. And we'll initialize `prevPt` to be that point. You will note briefly that if it were actual Java object, `prevPt` might be an error pointing at an object. But we're going to just write down the coordinates here to keep the diagram simple and legible. Next, we're going to do the steps for each point. So, we need to start at the first point, which is this one at the top of the diagram. And that will be the initial value of `currPt` as we enter the for each repetition. Then, we'll find the distance between these two points, which is 10. And we'll update `totalPerim` to be 10, 0 plus 10. And then, we'll update `prevPt` to be `currPt`. Now we're at the end of our for each repetition. So we'll update `currPt` to be the next point in the shape, which is (-3, -4). When we update `currPt`, we go back to repeat these steps. We repeat the steps again. Find the distance between those two points, which is 8. And then we update total perimeter to be 10 plus 8, or 18. And then we update `prevPt` to be the current point. Then we go back to the top of our loop, updating `currPt`. We repeat these steps for the last point in our shape. After which we've gone through all the points. So we skip to the steps after our repetition. Here, we can say that `totalPerim` is our answer. TotalPerim is 24. Is that the answer you came up with earlier? Yes, that is the perimeter of this shape. The fact that our algorithm came up with the right answer here gives us more confidence that we generalized correctly. We're done executing the algorithm by hand and we have some great confidence that it's correct. So we're ready to turn the algorithm into code.

Video 4. Seven Steps in Action: Translating to Code

A screenshot of a Java IDE window. The window has a menu bar with 'Compile', 'Undo', 'Cut', 'Copy', 'Paste', 'Find...', 'Close', and a 'Source Code' dropdown. The code editor displays the following Java code:

```
import edu.duke.*;

public class PerimeterRunner {
    public double getPerimeter (Shape s) {
        //Start with totalPerim = 0
        double totalPerim = 0;
        //Start with prevPt = the last point
        Point prevPt = s.getLastPoint();
        //For each point currPt in the shape,
        for(Point currPt: s.getPoints()){
            //Find distance from prevPt pt to currPt, name it currDist
            double currDist = prevPt.distance(currPt);
            //Update totalPerim to be totalPerim + currDist
            totalPerim = totalPerim + currDist;
            //Update prevPt to be currPt.
            prevPt = currPt;
        }
        //totalPerim is my answer
        return totalPerim;
    }
}
```

All right, now you've developed the algorithm to find the perimeter of a shape, and we want to turn it into Java code. You've seen a lot of Java code as we've talked about the syntax and semantics. And so what we're going to do is step by step go through and take the algorithm that we wrote, and turn it into code that has semantics that match what we wanted our algorithm to do. So here I have a class called Perimeter Runner, it's going to have the method getPerimeter, which takes a shape and return to double. It's got some other code in here, so it has a main that we can run, which will make an instance of this class and call testPerimeter, which will use FileResource to read it from a file and create a shape and call the perimeter and then print that out. We're going to go through and translate this to code, and we've written our algorithm here as comments, which are things for people that the compiler will ignore. So the first thing we say is, start with totalPerim equals zero. So this sounds like we need a variable, we called it totalPerim, we're going to set it equal to zero. We need to think about what type we want. Since we said it equals zero, we might want an int. But if we think a little more carefully, we might realize we want to double, since we're working with floating point numbers that might have fractions. So double, totalPerim equals zero. And if we'd written this as int we, hopefully, would find that out in testing, when we come up with answers that should have fractions and we don't have them. Then, we say start with previous point equals the last point. So we need another variable, previous point. What type of variable is this? Well, this is going to be a point, and it's going to be the last point. We meant in s even though we didn't write that down, so we'll call get last point. How do I know that was there? I looked at the documentation for shape before I started doing this video and saw that it has a getLastPoint method. Then, we save for each point, which we wanted to call curve point in the shape. So this sounds like a for each loop. Each point, curr point in the shape.getPoints, which is going to give us all of the point. We want to do these steps, and I'm going to go ahead and put them in curly braces. We want to find the distance from previous point to the current point and name it curr distance. So anytime we want to name a quantity, we want a variable. And so that's going to be the distance from prev point to curr point. Then, we're going to update totalPerim to be totalPerim, plus current distance. And then we're going to update previous point to be current point. Last we said, the totalPerim is my answer, whenever we know our answer, we're going to return that because that's how we give our answer back to whoever called us. So now that I've written this code, I'm going to come up here, and I'm going to click compile, and it says cannot find a symbol variable shape. That's because my shape was called s. And now it says class compiled with no syntax errors. So now, I'm going to shrink this window down a little bit smaller. I'm

going to come over here, and I'm going to do, main, even though that's how we run programs outside of BlueJ. If we've written a main, we can do that. We're going to give it no argument, so I'm just going to click okay, and it's going to ask me, "What input file has the points for the shape?". So, each of these files has some point. For example, example one has negative one three, negative one negative one, four negative one one three, which is what we used when we developed our algorithm. So that's a good first check, make sure we get the answer we expected. And that gives us 16, which you may recall is what we came up with when we worked this example ourselves. Of course, using that might be bad, because if we made a mistake, we wouldn't necessarily catch it, since we've already used those values in developing this. We might want another one, and so this says that this other shape with points of negative three, four, negative three, negative four and three negative four has a parameter of 24. If you work that out yourself, you'll find that that's the right answer. And so we become more and more confident that this code that we just wrote is correct. So that's how you turn your algorithm into code. You go through step by step, take each step, write down the code that corresponds to what you wrote.