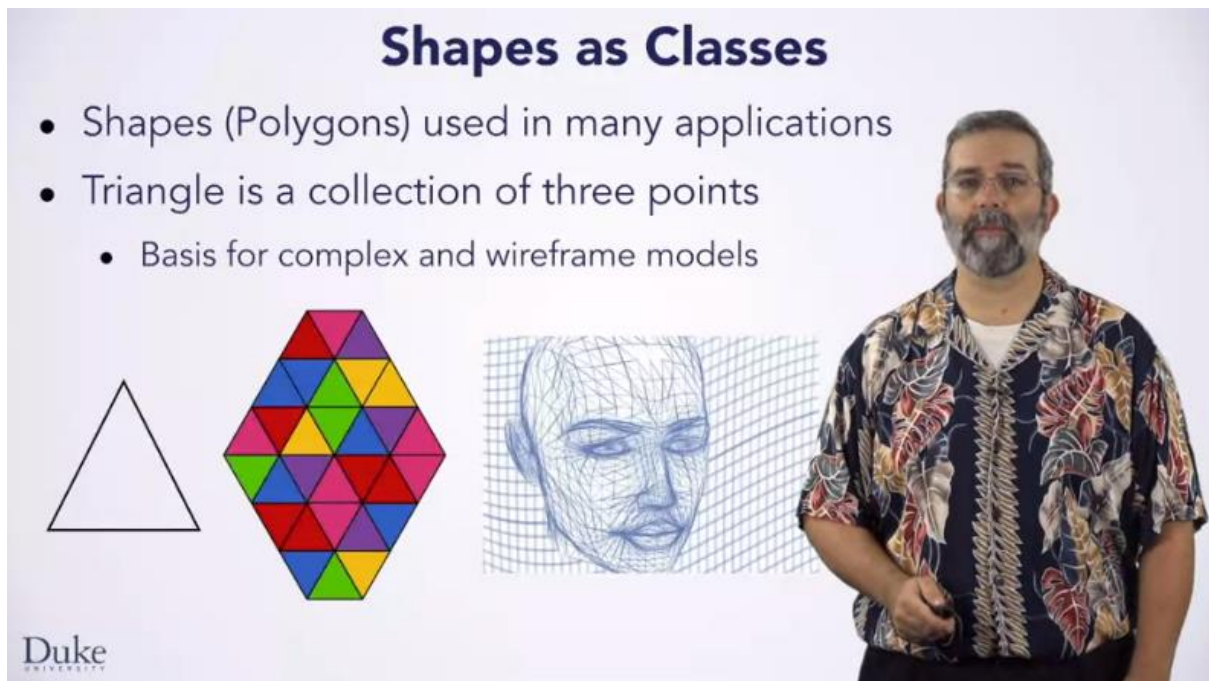


Variables and Mathematical Operations

Reading: Let's learn some basic Java syntax

Now that you are set up with BlueJ, you are ready to learn the basics of how to write code in Java-- that is, Java syntax. In addition, you will learn the meaning behind Java syntax, which will help you translate your solutions to problems into software. In the next few lessons, we will use a simple problem to help introduce you to Java, which Robert will introduce in the next video.

Video 1. Shapes: Collections of Points



The video thumbnail features a man with a beard and glasses, wearing a colorful Hawaiian shirt, standing next to a presentation slide. The slide is titled "Shapes as Classes" and contains a bulleted list of points. Below the text are three images: a simple triangle, a complex polygon made of many small colored triangles, and a wireframe model of a human head. The Duke University logo is in the bottom left corner of the slide.

Shapes as Classes

- Shapes (Polygons) used in many applications
- Triangle is a collection of three points
 - Basis for complex and wireframe models

Duke UNIVERSITY

Hello, we're going to learn how to calculate the perimeter of shapes using Java. We'll use what is called a **Java Class**, which you'll learn about soon, named **Shape** to model polygons in geometry. These kinds of shapes are used in many applications. For example, a triangle is the simplest polygonal shape, it's simply a collection of three points. But triangles can be combined into complex shapes that are colorful or form the basis of wireframe diagrams used in computer graphics and video games.

We'll use a Shape Java class to represent a collection of points. We'll use a class for point and for shape to understand programming concepts that are applicable across many programming languages.

Shape is Collection of Points

- Shape or Polygon is a collection of points
 - How do we construct a polygon/shape?
 - How do we access points in a polygon/shape?
 - What operations/methods can we implement?



Duke
UNIVERSITY



We'll need to construct shapes by adding points one at a time. This will allow us to create shapes like this six sided shape, which is constructed from six points. Here is a shape with five points. Here is another six sided shape. The order in which the points are added to the shape is important. When we look at the code for the Java Shape class, we'll also need methods for accessing the points, either one at a time or using other approaches. We'll likely want to create operations or methods that use shapes, like drawing a shape or finding the perimeter of a shape.

Shape Can Be Complex

- Shape can be simple or complex, comprised of just a few points or many, many points.



Duke
UNIVERSITY


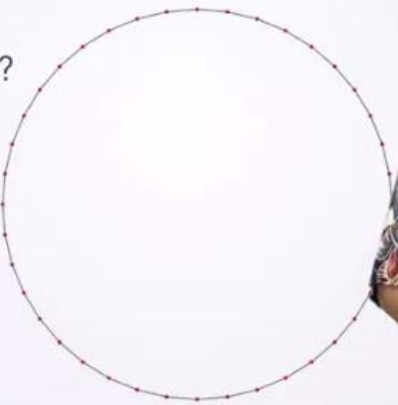


We've seen that shapes can have many points. Our shape class is simple, but we could expand it to create complex shapes of many, many points. Like this butterfly. When viewed more closely, it's clear

that the butterfly is simply a collection of colored triangles, like the wireframe drawing we saw earlier. The shape class we'll look at is very simple.

Shape is a Simple Class


- Circles and other shapes aren't a finite collection of points
 - What is a circle?
 - Can we model one?



Duke UNIVERSITY

Circles aren't really a finite collection of points. So circles and some other shapes are harder to model using our Java class. In geometry, a circle is an infinite collection of points that are all the same distance from the center of a circle. Our Java class is just a finite collection of points. However, even with our simple class, we could model a circle using many points, as shown in the diagram here. So, even our simple class will prove very powerful. After we cover some of the basics of writing programs in Java, we'll explore the point and shape classes in more detail.

Video 2. Why Semantics: Motivation to Read Code



The slide is titled "Understanding Code" in a large, bold, dark blue font. It contains a bulleted list of questions and concepts. To the right of the slide, a woman with grey hair and glasses, wearing a red short-sleeved shirt, is gesturing with her hands as if presenting.

Understanding Code

- Java classes Shape and Point
- What can these classes do?
- How would you the code on your own?
 - Need to know **semantics** of code
 - Fancy word for meaning
- What are semantics of code?
 - How to execute step-by-step by hand


Duke UNIVERSITY

Okay. You will be using the Java classes Shape and Point, but what will you be able to do with these classes? Well, in some ways, you know the answer to that. You will be able to draw shapes or calculate a shape's perimeter, but just knowing what a Java class or script could do does not mean you understand it. So the question we are going to answer now is, how would you understand what this code does by yourself? That is, what are the **semantics** or **meaning of each part of the code**? Understanding the precise meaning of code is important because you can't write code without saying precisely what you mean. When we talk about **understanding the semantics of code**, what exactly do we mean? We mean **how would you execute the code by hand with nothing but pencil and paper?** This skill is very important for a couple of reasons. First, it's how you understand code well enough to write what you mean. Second, when your code does not behave as you expect, how do you figure out what is going wrong? You need to understand what it does, and this gives you the skills to do that.

Video 3. Variables



Hello. We're going to start learning to execute code with the most basic statements in Java and build up to more complex statements from there. Here, we have two variable declarations. One for an int called X, and one for an int called Y. If we execute these statements as is, we will create boxes labeled X and Y to hold the values we eventually decide to put in these variables. But in this example, we have not explicitly provided any initial values. In some languages such as C, no default value is ever provided when you declare a variable. Meaning, you get undefined behavior if you use an uninitialized variable. This is such a common and significant problem that Java provides two solutions: an initial default value of zero is given to instance variables, or an explicit error is given for using a local variable before initializing it. We will see these different types of variables in upcoming videos.




```
int x = 4;  
→ int y = 6;
```

x	4
y	6

Duke
UNIVERSITY

Now, let us look at another example which initializes the variables when they are declared. Here, the first statement declares an int called X. Executing it makes a box for X and immediately assigns the value 4 to X. We show this by putting 4 in the box for X. Executing the next line, both creates a box for Y and puts the value 6 in that box. **Combining the declaration and initialization of your variables into the same statement won't pose a problem in any language.** And anyone reading your code knows exactly what you meant for the code to do. So this is a good habit to get into.



```
int x = 4;  
int y = x + 2;  
→ int z = y - x;
```

x	4
y	6
z	2

Duke
UNIVERSITY

Of course, variables are only useful if you make use of the value that they have. Here, the first code declares X and initializes it to 4. Next, Y declares and initializes it to X + 2. To perform this statement

we must evaluate the expression on the right hand side of the equal sign. X is 4, so $X + 2$ is $4 + 2$ or 6. We show this by creating a box for Y and putting 6 into it. The last statement creates a variable Z and initializes it to $Y - X$. So you would take the value of Z, which is 6, and the value of X, which is 4, and subtracts them to get 2. You would then create a box for Z and initialize it to 2. Great. Now, you know how to execute code with variable declarations and assignment statements.

Video 4. Mathematical Operators



```
int x;  
x = 4 + 3 * 2;  
int y = x - 6;  
→ x = x * y;
```

x	40
y	4

Duke UNIVERSITY

Now that you know a little bit more about expressions, let's see them in action. This code example starts with the declaration of an integer, or int variable, called x, which behaves exactly as you've already seen. Next we initialize x to $4 + 3 * 2$. As you know from math, times has higher precedence than plus. So this expression evaluates to $4 + 6$, which is 10. So we put 10 in the box for x. Next we declare another variable, y of type int. And initialize it to $x - 6$, which is $10 - 6$, which is 4. So we create a box for y and put 4 in it. The last statement says, x gets $x * y$. Sometimes novice programmers expect statements which look like this to behave like algebraic equations with an equal sign, where you might solve for x. However, that's not what happens. Instead, you follow the rules you've already learned. The right hand side evaluates to $10 * 4$, which is 40. And you put 40 in the box for x.



Now, let's see another example. Before we work through this, take a moment to pause the video. And see if you can figure out what values x, y, and z have at the end of this code fragment. Okay, let's step through it. First, we declare and initialize x. Next, we evaluate $x * 3$, which is 6 and initialize y to that value. Next we compute $y/2$, which is 3 and initialize z to that value. The last statement says x gets $(2+z)$ mod 2. Since the $2+z$ is in parentheses, we compute that first and get 5. Next, we compute 5 mod 2. Remember from your reading that 5 mod 2 means we divide 5 by 2, but take the remainder, not the quotient. So this expression evaluates to 1. So we update the value in x's box to be 1. Okay, great, now you should be able to evaluate code involving a wide variety of mathematical expressions.

Video 5. Functions

The next important idea to understand is function calls. **Functions** extract a computation out, giving it a name and parameters. You then can use the function to perform that computation without rewriting it. You can also think about what the function does and not how it does it. Technically speaking, Java doesn't have functions. It has methods since all code in Java is inside of objects. However, before we learn the more complex behavior of objects and methods, we'll learn the basic principles of function calls. These concepts will then lay the foundation for understanding method calls.


```

int myFunction(int x, int y) {
    int z = 2 * x - y;
    return z * x;
}
int f(int n) {
    return 3 + myFunction(n, n+1);
}
int g() {
    int a;
    a = myFunction(3, 7);
    int b = f(a * a);
    return b;
}

```

g	
a	0

1 myFunction	
x	3
y	7

1. Create **frame** for called function
2. Pass **parameters**: copy values into frame
3. Note where to return when finished ("**call site**")
4. Move execution arrow into called function

Duke UNIVERSITY

We have three functions here. My function, f and g. Why are we starting at g? We'll assume for now that you chose to call g from the BlueJ interface. Later, you'll learn about the main method which is where programs start when you run them outside of BlueJ. We start with a frame for g and with the execution right at the start of that function. The first line declares a, so we create it's box inside of the frame for g. Next, we're going to set a equal to the value of the function call myFunction(3, 7). To evaluate this expression, we need to create a frame of the function that we're calling. In this case, myFunction. This will hold the parameters and variables of myFunction. Next, we pass the parameters to myFunction. We create a box for each parameter with the names coming from the function declaration, x and y. We initialize these by copying in the values of the expressions that were passed. Here, 3 and 7. Next, we need to note **where to return** when we finish executing my function. This location in the code is named the **call site**. The place where the function was called. We'll note it with a marker one in the code and put the same marker in the corner of the frame. Finally, we move the execution into myFunction and start executing code there. Here, we declared an initial of z. Evaluating the expression 2 times x minus y. The values for x and y come from the frame from myFunctions, 3 and 7 respectively. So, z will be -1. Now, we have reached a **return statement**.

```

int myFunction(int x, int y) {
    int z = 2 * x - y;
    ➔ return z * x;    -1 * 3 =
}
int f(int n) {
    return 3 + myFunction(n, n+1);
}
int g() {
    int a;
    a = myFunction(3, 7);
    int b = f(a * a);
    return b;
}


```

1. Compute **return value**
2. Find where to return (noted in frame)
3. Copy return value to call site
4. Move execution arrow back to call site
5. Destroy frame

Duke
UNIVERSITY

g	
a	0

1 myFunction	
x	3
y	7
z	-1



Return statements tell us to leave the correct function, returning to the call site noted in the frame. They also tell us the value to return to the caller. The first thing we need to do is evaluate this expression to obtain the return value. Here, the expression is $z \text{ times } x$. So we evaluate -1×3 and we get -3 . Next, find where we should return. This is the call site we noted in the frame, then we copy the return value back to the call site. The function call evaluates to this return value. We move the execution arrow back to the call site and destroy the frame for the function we just returned from. Now, we're back in `g`. The call to my function evaluated to -3 . So, this line behaves like `a gets -3`. We'll finish that assignment, putting -3 in its box. Our next line again has a variable declaration and a function call. We make a box for `b` and go through the same process to call `f`. You make a frame for `f` and pass parameters. This time, there's one parameter `n`, whose value is `a times a` which is 9 . We know where to return and begin executing code inside of `f`. Our next statement is a return statement, but the expression involves a function call. So, we have to evaluate that call before anything else. We start with a frame and pass parameters. `x` gets a value of `n`, which is 9 . And `y` gets a value of `n+1`, which is 10 . We know the call site will use two this time since we are already using one somewhere else and move the executioner to the start of `myFunction`, and start executing code there. We declare `z` and initialize it to `a`. Now, we are ready to return from my function. We evaluate `z times x` which is 72 , then we find the call site noted in the frame and copy the return value 3 . Finally, we move our execution arrow back to the call site. Destroying the frame from `myFunction`. Now, we pick up where we left off in `f` using 72 for the value of the call to `myFunction`. We evaluate $3 \text{ plus } 72$ to get 75 . Since we're evaluating the return statement, this is the return value of `f`. So we find the call site and copy the return value there, then we return to that call site. Destroying the frame for `f`. Now, we can finish the initialization of `b`. `B gets 75`. Lastly, we reach the return statement for `g` which is where we started. When we return from the function where we started, we are done.

Video 6. Conditionals

```
int f (int x, int y) {  
    if (x < y) {  
        System.out.println("x < y");  
        return y + x; 4 + 3 = 7  
    }  
    else {  
        System.out.println("x >= y");  
        if (x > 8) {  
            return y + 7;  
        }  
    }  
    return x - 2;  
}  
  
int g () {  
    int a = 1 (3, 4);  
    int b = f (a, 5);  
    return b;  
}
```

g	
a	0

1 f	
x	3
y	4

Output
x < y

Duke UNIVERSITY

Now, let's see how to execute code with conditional statements in it. Here we have a function `f`, which has some **if** and **if, else** statements. We have another function `g`. We'll assume we've used BlueJ's interface to invoke the method or function `g` to start with. The first statement declares a variable `a`. So we'll make a box for `a`. Even though this statement initializes `a`, it's going to take us several steps to compute that value. So we're going to leave `a`'s value as 0, until we compute the value of the method `f`, applied to arguments 3 and 4. To evaluate this called to `f`, we make a frame, passing the values of the parameters. Note where to return and then we move the execution arrow to the first line `f`. The next line of code is an **if** statement, whose conditional expression is `x` less than `y`. We evaluate that expression and find 3 less than 4 is true. So I move the execution arrow into the then clause of this **if** statement and continue executing. The next statement is a call to **System.out.println** which is how you print something in Java. We write `x` less than `y` in our output and then, we return `y` plus `x`. This values follows the same rules you've already learned for return statements. We evaluate the expression to get the return value which is 7. Which is then what the function call evaluates to and will return back to the caller, destroying the frame for `f`. Now we're ready to finish initializing `a`, since we know that the call to the method `f` evaluated to 7. So, `a` is now 7.

```

int f (int x, int y) {
    if (x < y) {
        System.out.println("x < y");
        return y + x;
    }
    else {
        System.out.println("x >= y");
        if (x > 8) {
            return y + 7;
        }
    }
    return x - 2; 7-2 = 5
}

int g () {
    int a = f (3, 4);
    int b = 1 (a, 5);
    return b;
}

```

g	
a	7
b	0

1	f
x	7
y	5

Output
x < y
x >= y

Duke UNIVERSITY

Now, we're ready for the next line of code in g. We make a box for b. And we call f passing in the arguments 7 and 5. Once again, we want to evaluate the condition x less than y. However, now x is 7 and y is 5. So x less than y is false. We find the closed curly brace for this if statement and see that the if statement has an else clause. So we move the execution arrow into the start of the else clause and continue executing from there. First, we see the System.out.println statement and this prints the line x is greater than or equal to y. Then we reach another if statement. This if statement is nested inside of the else but that doesn't affect the rules of how we evaluate it. We see that the conditional expression is false since 7 is not greater than or equal to 8. There's no else clause, so we move the execution arrow immediately past the closed curly brace and continue execution. **There are not anymore statements inside the else clause**. So remove the execution arrow **outside of the else clause** and keep going. Next, we have a return statement. So we evaluate the expression x minus 2 to find that the return value is 5. This then gets returned to the place we noted. So we destroy the frame for f and return back to g. We finished the assignment statement and now we're ready for the return statement from g. We execute that and we're done with method g.