


Classes, Types, and For Each Loops


Video 1. Classes

As you continue learning how Java works, it is important to talk for a bit about objects and classes. Before we delve into the specific semantics, let's talk for a minute about the high level concepts. When you are writing a program, you often have data in the form of variables, which store the values you are computing on. And code, when it manipulates that data according to the algorithm you have designed. **Object-oriented programming** is a paradigm of programming languages that groups data, and the code that manipulates it, together into logical units called objects. This language design aims to help programmers think about their program by grouping the **code and data together into one logical unit**. As you write larger and larger programs, this principle becomes more and more helpful. As you progress and learn more about Java, you will learn about a lot of important principles that go along with these ideas. For now, however, we're just going to start with the basics.

```
public class Point {  
    private int x;  
    private int y;  
    public Point(int startx, int starty) {  
        x = startx;  
        y = starty;  
    }  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public double distance(Point otherPt) {  
        int dx = x - otherPt.getX();  
        int dy = y - otherPt.getY();  
        return Math.sqrt(dx * dx + dy * dy);  
    }  
    public static void main(String[] args) {  
        Point p1 = new Point(3,4);  
        Point p2 = new Point(6,8);  
        System.out.println(p1.distance(p2));  
    }  
}
```

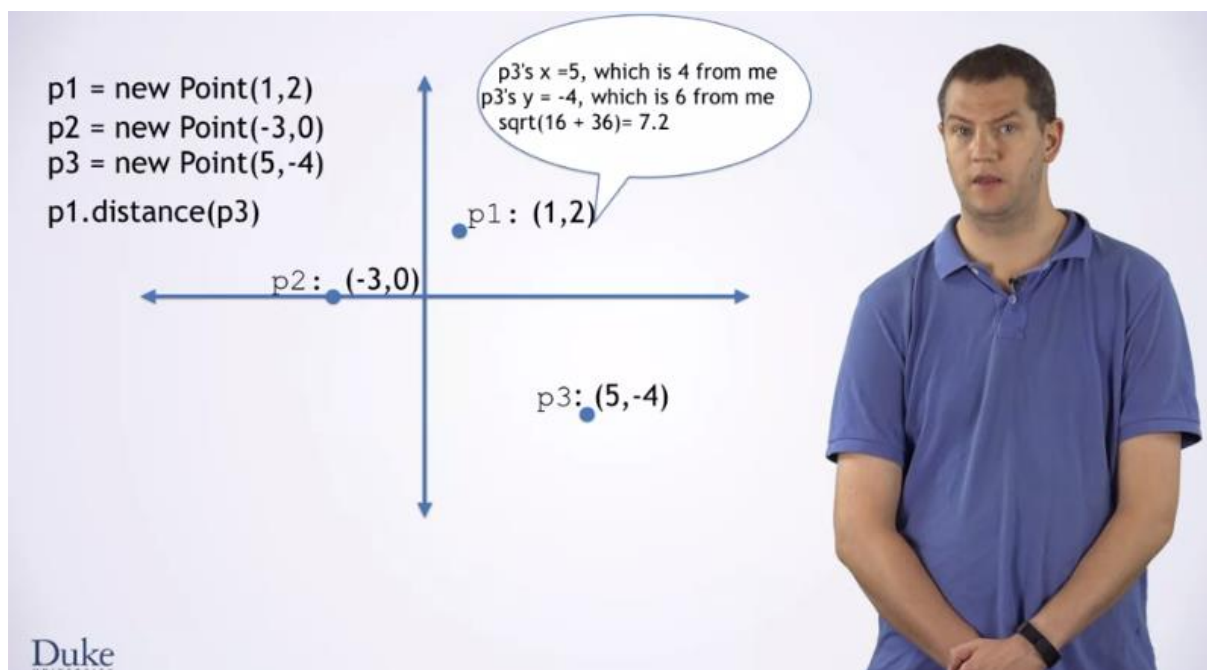
A class is a template
to make objects





Here, you can see an example of a class declaration. A class is a template that specifies how to make objects. Let's look at each piece of this declaration. (1) The first line tells Java that we are declaring a class, called Point. As with variables, we have a lot of freedom in what we name classes that we create. But we should name them descriptively. Here, we are making a class which represents a two-dimensional point, so Point is a good name for it. (2) Next we declare two fields, int x and int y. Field is the name for a variable that is inside of an object. They're also called instance variables since they are variables that are in each instance of the objects created from the class. These look like variable declarations except that the word private comes before them. **Private** means that **only code inside of this class can directly manipulate these fields**. You'll learn more about why that is important as you become more skilled in Java, but for now we'll just make all fields private. (3) Next is the declaration of a **constructor** for the class. A constructor **specifies how to create objects of this class**. It is code that gets run when an object is created to initialize that object. Note that the constructor looks like a function but has no return type. And is named the same as the class. These are the hallmarks of a constructor declaration. In front of the constructor, we have the word public, which means that any code

can use this constructor to create a point. (4) After the constructor, we have **three methods**, getX, getY and distance. Methods are **functions that are inside of classes**. In Java everything is inside of a class, so technically all functions in Java are methods. These methods are invoked or called on a particular object and implicitly act on that object. You can see a method call here where the code says **otherPt.getX**. This calls the get X method on the other point object. It will get the X of that particular point object. (5) Last, we have the Declaration of a **static method**. These behave a little differently from regular methods. They don't act on any particular instance of a class, they just belong to the class in general. That concept is a little tricky and we'll explain it more later. This method is called **main**, which is a special method. If you run your programs outside of Blue Jay, main is the starting point. Execution begins in main before any objects are even created.



The ideas of objects are supposed to help programmers think about their data in terms of objects that make logical sense. For example, if we make a new point, we are creating an object which represents something we can concretely think about, in this case a point on a plane. We can then make another point which has its own X and Y and represents a different instance of the same type of thing, another point on the plane. You can, of course, create as many of an object as you need for your algorithm.

Once you have some objects, you can invoke methods on them, such as `p1.distance(p3)`, which you can think of as asking p1 to compute the distance to p3. That is, you can think of this line of code as saying p1 go figure out how far you are from p3. You can think of the code that executes for this method call as logically belonging to the p1 object.

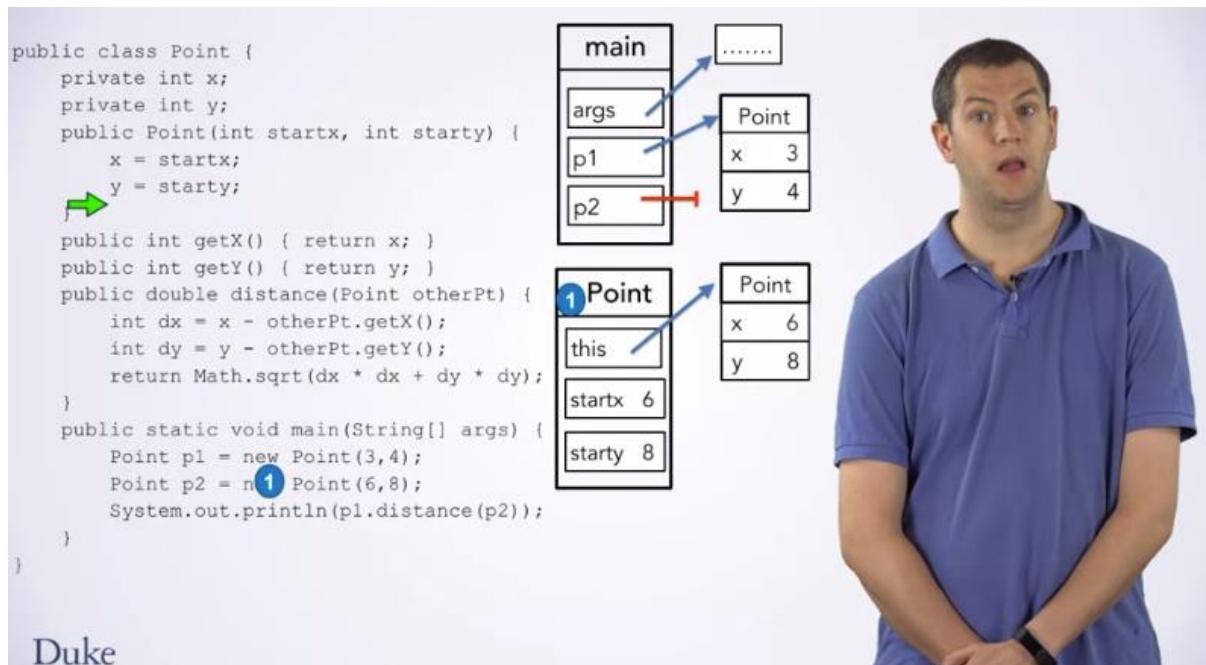
Video 2. New

```
public class Point {
    private int x;
    private int y;
    public Point(int startx, int starty) {
        x = startx;
        y = starty;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public double distance(Point otherPt) {
        int dx = x - otherPt.getX();
        int dy = y - otherPt.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
    public static void main(String[] args) {
        Point p1 = new Point(3,4);
        Point p2 = new Point(6,8);
        System.out.println(p1.distance(p2));
    }
}
```

The diagram illustrates the execution of the `Point` class constructor. It shows the `main` method frame, the `Point` constructor frame, and the `Point` object being created. The `main` frame has variables `args` and `p1`. The `Point` frame has a `this` pointer and variables `startx` and `starty`. The `Point` object has fields `x` and `y`, both initialized to 0. Arrows show the flow of execution and data.

Now that you have the high level concepts, let us delve into the step by step execution of our example `Point` code. We're going to start in `main` which we'll talk about more later. `Args` gives you access to the command line arguments which you won't need for quite a while. So we're just going to ignore that. The first line declares point `P1` and initializes it to a new point. Note that classes are types, so we can use class types that we make to declare variables. Here we want `P1` to be a point. Until we finish evaluating the right hand side, we're going to draw this as an arrow with a flat end. We'll use this notation to indicate when a variable does not reference an object. Note that this is a bit different from numeric variables like ints whose initial value is zero. We've colored this flat ended arrow red to remember that we have not yet explicitly initialized this variable. Now, let us evaluate the right hand side of the initialization expression. We're doing **new** which will create a new object. What type of object are we going to create? We'll look right after `new` and see that we are creating a new point. So we're going to draw a box for a point which has fields `x` and `y`. Note that the box we just drew is not in the frame but is outside of it. It is in a different area called the **heap**. Anytime you use new, you create data in the heap. The important difference is that data in the heap **does not go away when a function returns destroying its frame**. Note that we have put zero in the field of this new point and colored them red as we have not explicitly initialized them yet. Speaking of initialization, remember that we said that the whole point of a constructor is to initialize a newly created object. The next thing that happens is that we call the constructor to initialize this point. As with any call, we set up a frame and pass parameters. However, constructors and methods take an additional implicit parameter which tells them which object they are operating on. That parameter is called `this` and its value is an arrow which points to the object that is being acted on. In this case, `this` points to the object that we are creating to tell the constructor which object to initialize. Now, we enter the code for the constructor and begin executing statements there. The first line says `X` gets `startX`. There's no `X` in this frame, so where do we store the value of `startX`? Here `X` refers to the field inside of this object. That is, we want to put `startX`'s value into the `X` field of the object we're initializing. So to find the right box we follow the arrow for `this` and then look for the `X` field in this object and store the value three into that box. On the next line, `y` again refers to the field inside of this object. So we update that field to be four. Now we have finished the code inside the constructor and are ready to return to `main` at call site one. Back in `main`, we need to finish this

assignment statement. To do that, we need to store the value of the right hand side into the box for P1. A new expression evaluates to an arrow pointing at the object that it created. So we'll make P1 to point at the newly made point.



The next line does a similar thing. It declares P2 and initializes it to a new point. So once again we create a box for P2. We haven't initialized it explicitly yet. So we have a default value of a flattened arrow meaning it does not point to any object yet. We color red to remember that it has a default value. We then create a new point with its x and y fields set to default values of zero. And we call the constructor to initialize this point. Notice how this points at the newly created point. With multiple points in our world, it is important that we can keep track of which point we are working on. We then initialize the x inside of this point to be six as before we found the right box by following the arrow from this and we initialize the y inside of this point to be eight. Now we have finished the constructor and are again ready to return to main. In main, we finish the assignment statement setting P2 to point at the newly created object. We'll stop there and pick up with executing the method call in the next video.

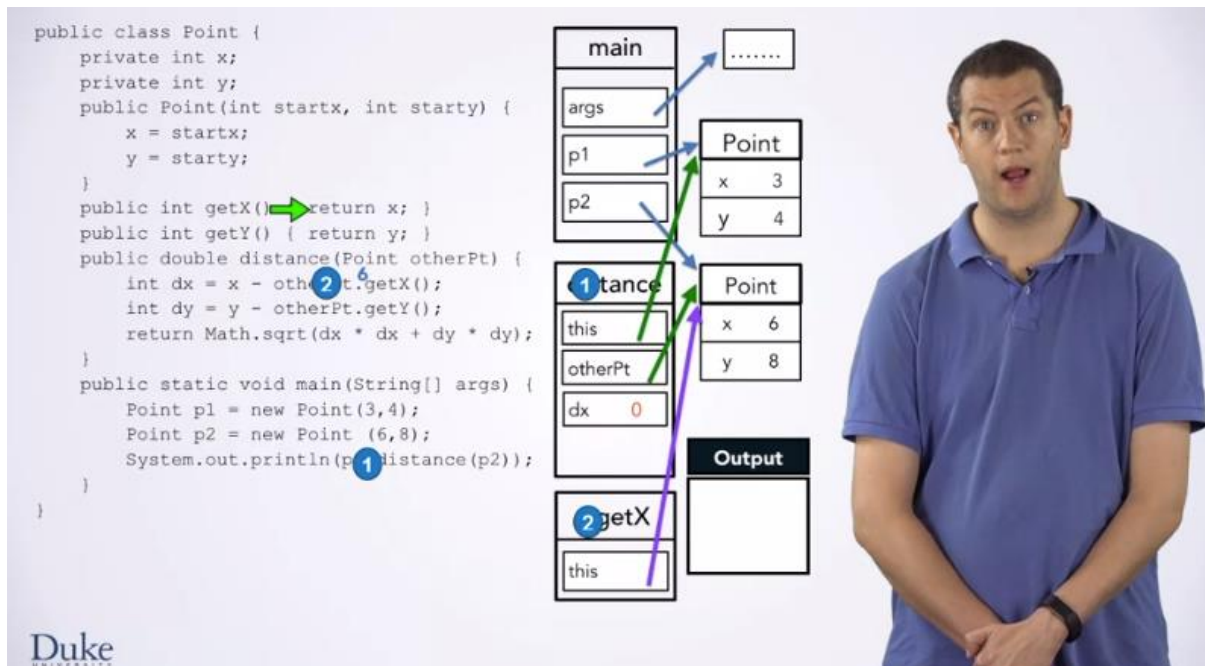
Video 3. Methods

In the previous video, we executed the declaration and initialization of points p1 and p2. And you learned about new in constructors. Now, it is time for you to learn about **Method Calls**. Method Calls work a lot like Function Calls, except that we have to pass this parameter to let the method know which object it is working on. Let us resume where we left off. This line is going to call p1.distance p2 and then print its return value.

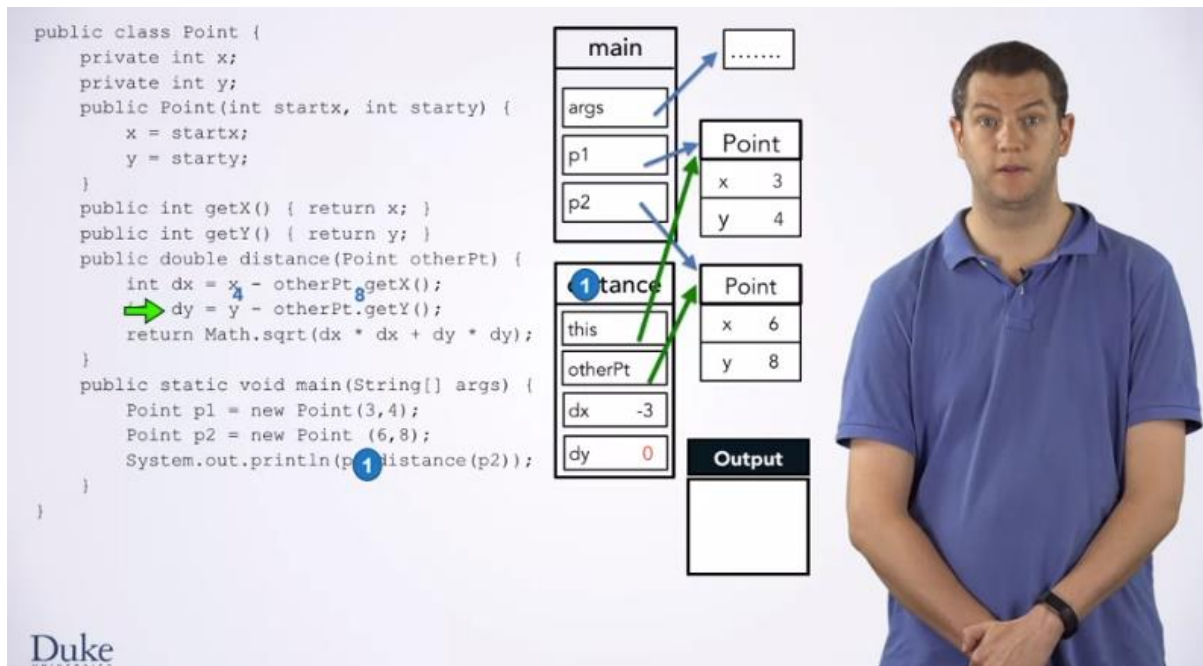
We need to set up a frame for distance which will take two parameters. The implicit **this** parameter which tells it which object it is acting on, and the **otherPt** parameter, which is explicitly passed to it. The this parameter has the same value as the variable before the dot. In this case, we have p1.distance, so this has the same value as p1. It is an arrow pointing at the same Pobject. We have colored this arrow differently but that does not have special meaning. It is just to help you keep straight which arrow is which in this diagram.

For the otherPt parameter, we just copy the value that is passed in. In this case, p2. So the value of other

point is an arrow pointing at the same point object as p2.




Now, we go into the `distance` method and begin executing code there. Executing the first line, we are declaring a variable `dx`, and need to initialize it to `x - otherPt.getX`. To evaluate that expression, we need to call **`otherPt.getX`**. So, we need to set up a frame for that method call. Again, this method takes an implicit `this` parameter to tell it which object to do `getX` on. Which object should this point at? Well, we did `otherPt.getX`. So, we copy the value of `otherPt` which is an arrow pointing at that point object. Now, we go inside of `getX` and begin executing code there. Here, we have `return x`. So, we need to evaluate the expression `x`, and return its value. How do we get the value of `x`? We follow the `this` pointer to find the object we are working on and then, look for the `x` field inside of that object. The value of that field is six. So, that is the value of the expression `x` here which is what will return to the caller at CallSite two. Returning to CallSite number two, we want to evaluate `x - 6`. How do we evaluate `x` here? We again look at the `this` pointer which refers to this `Point` object, and we get the `x` field inside of that `Point` object. That value is three which is the value of `x` in this expression. So we will compute three minus six and initialize `dx` to minus three.



Now, we are going to declare and initialize `dy` using a very similar process. First, we make a box for `dy`, then we call `getY` on other point. Notice how this, is a copy of the value of other point, an arrow pointing at our second point object. We get the value of the `y` field out of this object which is eight, and return it to CallSite two where we return the execution arrow to. We evaluate `y` by looking in this object and finding its `y` field which is four. Now, we are ready to finish the initialization of `dy` to four minus eight which is negative four.

Our next line has a little bit of math. Let's look at it in detail. We are calling `math.square root of d_x squared plus d_y squared`. First, we can compute the value of the parameter which is 9 plus 16, which is 25. So we need to evaluate `math`, that square root of 25. This looks like a method call, but where did the `math` object come from? It turns out that `math` is a class not an object, and is part of the Java library. This is a **static method call**. The method is being called on the class in general, not on any particular object. Here, the `math` class is just a handy place to put a bunch of mathematical functions together. Since we don't have the code for `math.square root`, we have to know what effect it has. If we did not know, we would look it up in the Java documentation. However, as you might have guessed, this method just computes the square root of its argument. So `math`, that square root of 25, will return 5.0. Our `distance` function returns that value to its caller. So the value of the call to `distance` in `Main`, will be 5.0. We then return execution domain where we are ready to complete the print statement, which will print out 5.0. Now, we are done with `Main`. So we will return from it, destroying its frame and exiting our program.

Video 4. Types



Types: Interpretation + Operation


- Everything is a number!
 - Specifically, represented as **bits**
 - But not all numbers mean the same thing
- Types specify
 - **Interpretation**: what does a specific number mean?
 - **Operations**: what can you do?
 - And how should it be done?

Duke UNIVERSITY

At this point, you have seen a variety of types such as `Int`, `Point` and `FileResource`. But, what exactly is a type? A **type** specifies how data should be represented, interpreted, and operated on, as well as, what operations you can do with it. One important rule of computing is that, everything is a number. If you didn't learn about the everything is a number principle, and whatever intro courses brought you here, we'll give you some links to videos about it. Specifically, this means that everything is stored in the computer's memory as bits, ones and zeroes. But, not all numbers mean the same thing. Some numbers might mean plane numbers, while others might mean letters, and others might mean the locations of data in the computer's memory.

So, the **type** of a value specifies how to interpret those numbers. It tells Java how to assign meaning to the ones and zeros stored in memory. The type also specifies what **operations** you can perform on the data. The type tells Java not only what you can do but how it should be done. Let's talk about both of these points in more detail.

Interpretation



The diagram on the left shows a box labeled 'someFunction' containing a smaller box labeled 'x'. To the right of this box is a vertical stack of 10 bits. The 4th bit from the top is highlighted in blue. The bits are: 011010110101010101110101, 000000000000000000000000, 101010101011011000101010, 010010011001011000000010, 000000000000000000000000, 101101101110001010101010.

- If we looked in memory, we'd find 1s + 0s
 - They represent the data
 - But what do they mean?

Duke UNIVERSITY

We just said that the type tells Java how to interpret the ones and zeroes in memory. Let's talk about that a little bit more, huh? On the left, we have the conceptual representation of the program state that we have been working with. There's a box for a variable called `x`. On the right, we have a bunch of bits from the computer's memory. The blue ones correspond to `x`. But what do they mean for the value of `x`? Well, if `x` is an `int`, then these bits would mean it has a value of 1234567890. We're not going into the details of how you can figure that out here, nor do you need to know it for beginning Java programming. But, if you take a computer organization class, you will learn a lot more about how data is represented. The point I want to make here is that, if `x` had a different type, like float, then the same bits would have different meaning. These same ones and zeros would mean 1228890.25. And if `x` were a string, then the bits would be the location in the computer's memory of the actual string object, which would have a sequence of characters. Those would also be stored with a bunch of bits that would be interpreted as letters, since your type would be `char`.

Operations

- Types also specify what operations
 - Consider: $x + y$
- If x and y are both `int`:
 - Legal: does integer addition
- If x and y are both `String`:
 - Legal: does String concatenation
- If x and y are both `Points`:
 - Not legal: `+` is not a valid operation on `Points`

Duke
UNIVERSITY



We also said that the type tells us what operations we can do and how they are done. Consider this simple bit of code, `x + y`. Is it legal? And if so, what does it do? To answer this, you need to know the types of both `x` and `y`. If `x` or `y` are both `ints`, then this code is legal and performs integer arithmetic. If `x` and `y` are both `strings`, then this code is also legal but performs string concatenation. It makes a string with the letters of `x`, first, then the letters of `y`, immediately after. Notice that, even though the plus operation is legal for two different types, we may perform that operation differently for one type than for the other. If `x` and `y` are both `points`, then this code is not legal.

Conversion Between Types

- What if you need to convert between types?
- Some types can be converted **implicitly**

```
int x = 3;
double d = x; //implicit conversion
```
- Some types require an **explicit cast**

```
double d = 3.14;
int x = (int)d;
```
- Others require method calls

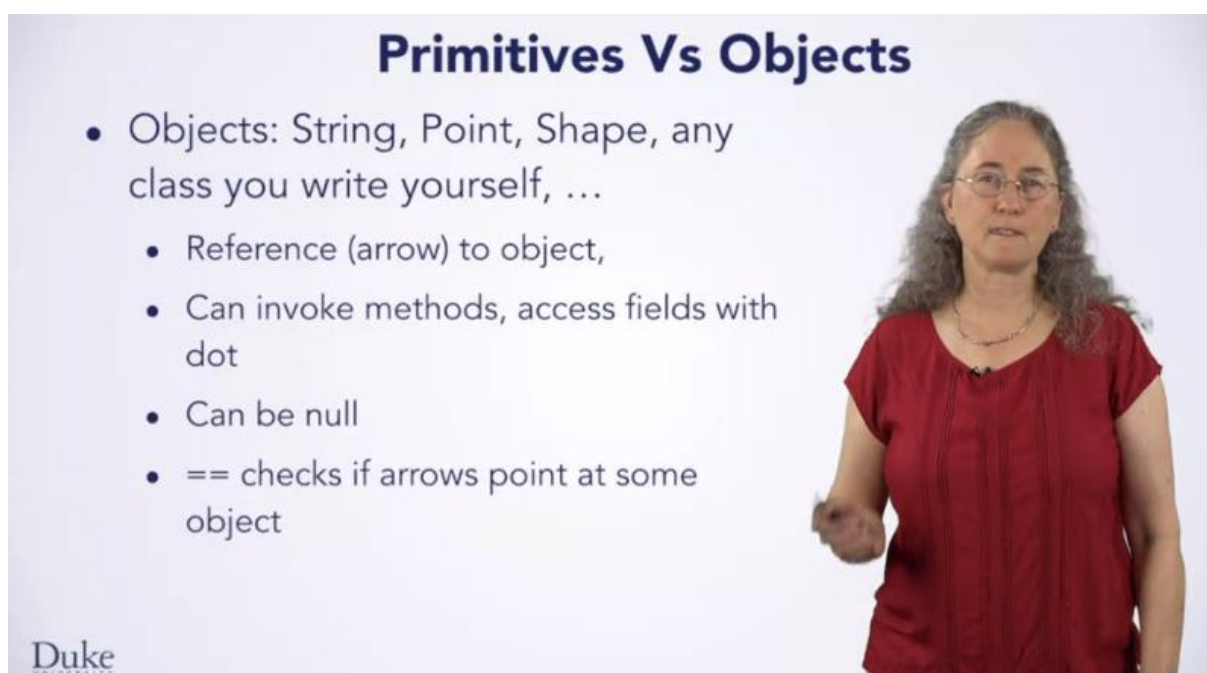
```
String s = "3";
int x = Integer.parseInt(s);
```

Duke
UNIVERSITY



While we're talking about types, you might be wondering what you'd do if you need to convert between types. The answer is that, it depends. For some type conversions, they can happen implicitly. If you

have an int and you need to convert it to a double precision floating point number, the compiler will just automatically insert the conversion for you without complaint. The general rule is that, you can use implicit conversion whatever the compiler will consider it safe. Here, we are turning 3 into 3.0, which is not a problem. Note that the compiler does not consider the values, only the types, when deciding if an implicit conversion is okay. For some type conversions, you can explicitly cast. This means, that you tell the compiler that even though what you're doing is questionable, you are sure you want to do it. Here, we are turning the double 3.14 into an int, which will discard the fractional part, leaving us with $x = 3$. The compiler wants to be sure that we meant to do this, so we explicitly cast by writing int in parentheses. Other conversions require calling methods to calculate out the converted value. For example, if we have this string "3", and want to turn it into an integer, we can't just directly cast it because the conversion is actually somewhat complicated. Instead, we have to call a method like Integer.parseInt which will perform the conversion.



Primitives Vs Objects

- Objects: String, Point, Shape, any class you write yourself, ...
 - Reference (arrow) to object,
 - Can invoke methods, access fields with dot
 - Can be null
 - == checks if arrows point at some object

A woman with grey hair and glasses, wearing a red top, stands to the right of the slide, gesturing with her right hand.

Duke
UNIVERSITY

The last thing we will mention about types is that, there are two major categories of types in Java, primitives and objects. There are eight primitive types: int, double, char, boolean, long, float, byte, short. We'll primarily use the first four of these. Variables of primitive types hold their value directly in their box. Primitive types don't have methods, so you can't do dot method call on a primitive, and they can't be null. Although, each primitive type has an associated wrapper class, which gives you an object to hold that primitive. Everything else is an object type. Some are built in the Java, like string. Others are part of libraries that you might use, like point or fileresource. And yet, others are classes you will create yourself. Whenever you make a class, the class you make is its own new type. Unlike primitives, the value of variables of object types is an arrow pointing at the object. This arrow is called a reference. You can invoke methods on the object with dot method name, and the reference can be null. Meaning, it does not refer to any object. If you do == on two objects, you're checking to see if the arrows point at exactly the same object. Okay, that's the basics of types. We know it's a lot to absorb at once. But, you'll get better with these ideas as you practice more Java.

Video 5. For Each Loops

Hi, now we're going to learn how a **for-each loop** works. This piece of code looks similar to the hello around the world example that you started with earlier. To make this work, we need to add an **import statement** to the top to tell Java where to find the `FileResource` class. `FileResource` is in a package that we provide to you to let you manipulate data in files before you learn the more advanced techniques and concepts that would let you do this in Java directly without the classes we've created. Accordingly, this is found in the `edu.duke` package. If you want to run this code in BlueJ, you could do so by creating a `HelloWorld` object. And then invoking its `runHello` method. We could also add a `main` method, which does the same thing if we want to be able to run the code directly, but outside of BlueJ, in another environment.

Let's start executing the code by hand. We start in `main` with a frame that contains `args`. We're not going to use this **args** argument, so we're not going to worry about its value. The first line declares **hw**, and **initializes it using new**. You learned about `new` in a previous lesson, but there's a slight difference here. The `HelloWorld` class does not have a constructor, so what do you do? In this case, Java provides a **default constructor**, one with no arguments, that does nothing. So we execute this line as you're used to, but we don't call a constructor. We make an object. Class `HelloWorld` has no fields, so the object itself doesn't have any state or anything else we can see in it. Then we finish the assignment statement. Next, we call **runHello**, passing in `hw` as the **implicit this argument**. Inside of `runHello`, our first statement declares the variable, **f**. The next statement initializes `f` to a **new FileResource object**. This raises two questions.

Code With for Loops

```
import edu.duke.FileResource;

public class HelloWorld {
    public void runHello(){
        FileResource f;
        new FileResource("file.txt");
        for(String line : f.lines()){
            System.out.println(line);
        }
    }
    public static void main(String[] args){
        HelloWorld hw = new HelloWorld();
        hw.runHello();
    }
}
```

The diagram illustrates the state of the program. It shows a `main` frame containing `args...` and `hw`. The `hw` variable points to a `HelloWorld` object. Below the `main` frame is a `runHello` frame, which contains `this` and `f`. The `this` variable points to the `HelloWorld` object, and the `f` variable points to a `FileResource` object. An `Output` box is shown at the bottom.

First, what does a `FileResource` object look like? Meaning, what fields are there in the `FileResource` object? It turns out we don't actually need to know this precisely. The details of how a `FileResource` object works can remain hidden from us as long as we know what it does. This is an instance of the important programming principle known as **abstraction**. Second, what does the constructor for a `FileResource` object do? To find the answer to this question, we would consult the documentation for `FileResource`. Consulting documentation for libraries, sometimes called APIs, that you use in

programming is an important task. Since `FileResource` comes from the Duke Learn to Program libraries, we would look at the documentation website. And we would read about the constructors for a `FileResource` class. There are three constructors, and the one we want is the one in the middle. Since we're passing a string literal in, the string, "file.txt". This documentation says the constructor will find that file with that name on our computer. Given this information, we're just going to represent the `FileResource` object as knowing what file we asked for, the file whose name is "file.txt". We don't know precisely what fields are stored in this object, but that's okay. We'll represent as precisely as we can because we know what the object does. We finish the assignment statement, so now `f` references this `FileResource` object we just created by calling the constructor.

Code With for Loops

```
import edu.duke.FileResource;

public class HelloWorld {
    public void runHello(){
        FileResource f;
        f = new FileResource("file.txt");
        for(String line : f.lines()){
            System.out.println(line);
        }
    }
    public static void main(String[] args){
        HelloWorld hw = new HelloWorld();
        hw.runHello();
    }
}
```

Output

Hello

The next line is new to us, It's a **for loop**. In particular, this for loop is often called a for-each loop because it does something for each value in an iterable. What's an iterable? The short and simple answer is that's an object which gives you a sequence of values. Let's dig a little more closely into the details of this loop. The first part declares a variable. The type is `String` and the name of the variable is `line`. This declaration will behave like any other, will create a box for `line`. Next is a `:`, this is the syntax of a for-each loop. We don't use an equal or a assignment operator since we aren't assigning one value to `line`. Instead, we're going to each value in the iterable sequence. Many people read the colon as **in** when they read the code out loud or to themselves, for `String line in f.lines`. Next, we have an expression which evaluates to the iterable whose values we want variable `line` to refer to one after the other. In this case, that expression is a call to `f.lines`. To understand what this does, we'll need to understand what `f.lines` returns. So we'll, again, need to consult our documentation. From this API documentation, we see that **`f.lines`** gives us an iterable whose values are each line in the file in the order that they appear. This means that we'll need to look at the file whose name is `file.txt` to see what its contents are in order to understand and simulate the code behavior.

On my computer, I've made file.txt, and put these two lines in it. Hello on one line and World on the next line. It isn't a very exciting file, but we want this example to be relatively short. Returning to our code, we now create this sequence of strings, Hello and World. We'll create the box for line. And we'll make it refer to the first element of this iterable sequence. Now we go into the body of the for loop and begin executing statements there. The next line is a print statement. So we print out the value of line, which is Hello. Now our execution arrow is just before the closed curly brace of the for-each loop. We've reached its end. When you reach the end of a for loop, you move the execution arrow back to the start of the loop to do the next iteration, the next time through the loop, with the new value for the loop variable. In this case, our loop variable is the String line. So we need to update it to have the next value in the iterable. We update line's arrow to point at the next value in this sequence. And again go into the body of the loop. We encounter our print statement, but this time, line has a different value, so we print World. Now we've once again reached the end of the body of the for loop.

So we go back to the beginning, the start of the loop. We need to update line to refer to the next value in the iterable sequence. However, if we try to do that, we'll find that there are no more elements in the iterable sequence. We already used the last one, so instead, we exit the loop. We move the execution arrow past the body of this for each loop and begin executing code there. When we do this, the loop variable, in this case the string line, goes out of scope. That means it no longer exists, so we remove its box. Now we're ready to return from the method runHello, destroying its frame. Since the call to that method, runHello, was the last line in main, we also return from main, exiting the program.