

## Week 2 Module 1 – Finding a Gene in DNA

### Video 1. What is a String

Hi, I'm Dr. Raluca Gordan. I'm a professor in the Duke Center for Genomic and Computational Biology, and on the Department of Biostatistics and Bioinformatics. My work is strongly based on designing and using computational algorithms, programs, and tools, and I want to tell you just a little bit about that.

But first, I'd like you to think about the word strings. What does an orchestra conductor think about when she hears the word strings? What does a sailor think about the word strings? What does a pianist think about the word strings?

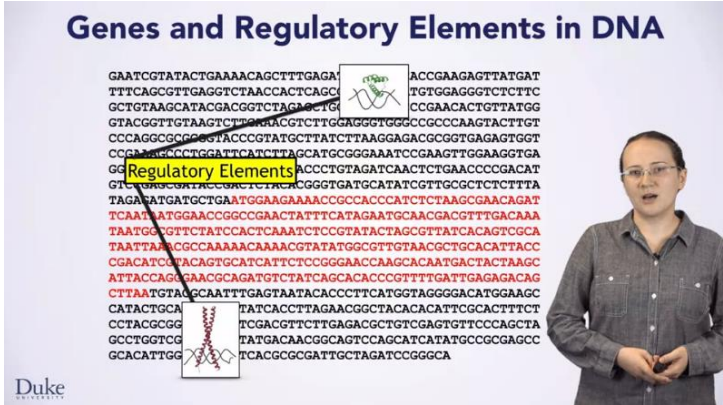
Well, I'm a genome scientist, so I will tell you what I think when I hear the word strings, and that is genomic strings. The genome of an organism stores all the genetic information necessary to build and maintain that organism. This genetic information is stored as a long list or string over the four letter alphabet A, T, C, and G. These four characters correspond to the four DNA bases, adenine, thymine, cytosine, and guanine.

The sheer size of the genome makes it difficult, if not impossible, to analyze by hand. The human genome, for example, contains 3 billion characters. That is 1 million times more than the characters shown here. Thus, finding any information in the genome requires computational approaches.

In addition, the genome is complex and contains different types of information. Computational approaches are needed to find information, including genes, as you can see here. Finding genes requires more than simply looking for the tags or codons that identify the start and end of a gene.

In addition to genes, such as the one shown here in red, we also need to look for regulatory elements, as shown here. We do this with computational tools and techniques. These regulatory elements are shown here as simple letters representing nucleotides. But it's important to remember that these are actually bound by proteins called transcription factors which help turn genes on and off.

**Genes and Regulatory Elements in DNA**



GAATCGTATACTGAAACAGCTTTGAGAACCGAAGAGTTATGAT  
TTTCAGCGTTGAGGTCTAACCACTCAGCGTGTGGAGGGTCTCTTC  
GCTGTAAGCATACGACGGTCTAGAGTCCGAACACTGTTATGG  
GTACGGTTGTAAGTCTGACACGCTTTGGAGGGTGGCCGCCAAGTACTTGT  
CCCAGGCGGGGTACCCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGT  
CCGATGACCTGGATTATCTTAGCATGCGGGAATCCGAAGTTGAAGGTGA  
GGTACCTGTAGATCAACTCTGAACCCGACATACCCGTGATGCATATCGTTGCGTCTCTTTA  
TAGATATGATGCTGATGGAAGAAAACCGCCACCATCTCTAAGCGAACAGAT  
TCAATATGGAACCGCGCACTATTTCATAGATGCAACGACGTTTGACAAA  
TAATGGGTTCTATCCACTCAAATCTCCGTATAGCTAGCGTTATCAGACTCGCA  
TAATTAACGCCAAAACAAAACGTATATGGCGTTGTAACGCTGCACATTACC  
CGACATCGACAGTGATCATTTCTCCGGGAACCAAGCACAATGACTACTAAGC  
ATTACCAGCAACGCAGATGTCTATCAGCACACCCGTTTGTATTGAGAGACAG  
CTTAATGTACCAATTTGAGTAATACACCCCTTCATGGTAGGGGACATGGAAGC  
CATACTGCAATACACCTTAGAACGGCTACACACATTGCGACTTTCT  
CCTACGCGGTCGACGTTCTGAGACGGCTGTGAGTGTCCAGCTA  
GCCTGGTCGTATGACAAACGGCACTCCAGGCATATATGCGCGAGCC  
GCACATTGGTCACGCGGATTGCTAGATCCGGGCA

Regulatory Elements

Gene

Duke University

My research is focused on identifying such regulatory elements in the human genome using various computational approaches. You will do similar things in this course and this will prepare you to become a computer scientist or a genome scientist depending on what your preferences are.

## Video 1. Understanding Strings

Hi, and welcome to this lesson on finding information and patterns in data. Which is a very general topic that we will make concrete, in working with Java strings.

### Why Work with Strings?

- Numbers are stored in a computer
  - In memory, on a flash-drive, on a hard-drive
- .mp4
- .png
- .txt

01001001	01000100	00110011	00000011	00000000	00000000
00000000	00000000	00000000	01010000	01010100	01000011
01001111	01001110	00000000	00000000	00000000	00000110
00000000	00000000	00000000	01000010	01101100	01110101
01100101	01110011	01010100	01000001	01001100	01000010
10001001	01010000	01001110	01000111	00001101	00001010
00011010	00001010	00000000	00000000	00000000	00001101
01001001	01001000	01000100	01010010	00000000	00000000
00000001	00101100	00000000	00000000	00000000	11001000
00001000	00000110	00000000	00000000	00000000	01010010
01010100	01101000	01100101	00100000	01000011	01100001
01110011	01101011	00100000	01101111	01100110	00100000
01000001	01101101	01101111	01101110	01110100	01101001
01101100	01101100	01100001	01100100	01101111	00001010
01000101	01100100	01100111	01100001	01110010	00100000

Duke UNIVERSITY

Strings are sequences of letters, digits, punctuation, any character that you might type for example. Why will you learn about strings? You've learned previously that everything is a number. That's true as you can see here, where I've captured the beginning of three different files. These files might be stored in memory, on a flash drive, or on your computer's hard drive. The first file, was a video. A file with a.mp4 suffix. The second file was an image, a file with a.png suffix. The third file was a plain text file with a.txt suffix. Can you tell which group of bits of zeros and ones goes with which file, by simply looking at the zeros and ones? Some people may be able to do that, but most cannot.

## Information Is Often Readable

- We use Strings to store readable-data
  - Readable by ... you, or a computer
  - Find patterns, knowledge, information

```
>JX477166v1
CGGACACAAAAAGGTTTTTAAGATTTTGTGTCGAGTAACTATGAGGAAGATTAAACAG
TTTTCTCAGTTAAGGTATACACTGAATTGAGATTCTCTCTTTGCTATTCTGTAACCTTCC
CTGGTTGTGACAATTGAATCAGTTTATCTATTACCAATTACATCAACATGGTATGCTAGTGATCTTG
GGACTCTTCTTCATCTGGTTTTCTTAGAGCTCTGAATCTATTTGTGAGAAGTTCATCCAAACGACCA

<div class="split-3-layout layout theme-base">
<h2 class="section-heading">
</h2>
<div class="column">
<article class="story theme-summary"

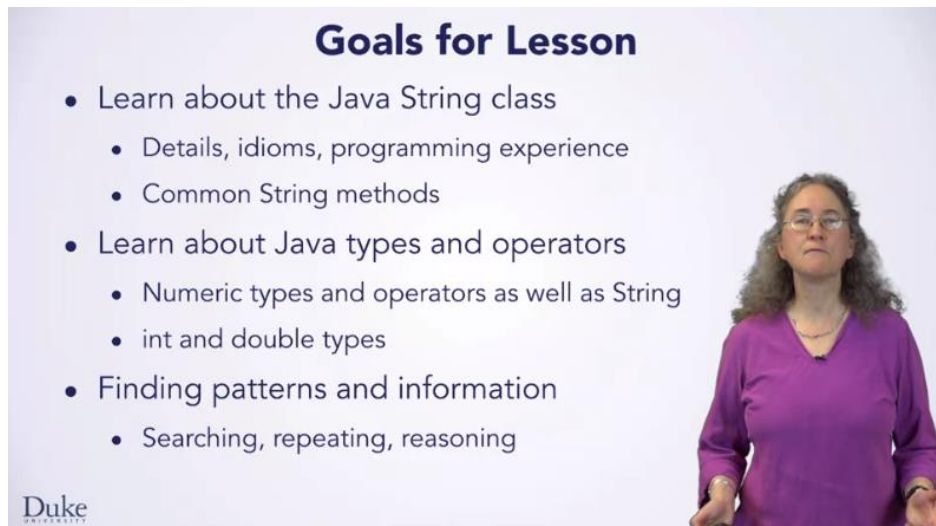
cdatetime,address,district,beat,grid,crimesdesc,ucr_ncic_code,latitude,longitude
1/1/06 0:00,2082 EXPEDITION WAY,5,5A,1512,***,2204,38.47350069,-121.4901858
1/1/06 0:00,4 PALEN CT,2,2A,212,***,2404,38.65784584,-121.4621009
1/1/06 0:00,22 BECKFORD CT,6,6C,1443,***,2501,38.50677377,-121.4269508
```

Duke  
UNIVERSITY



Although everything stored on a computer as a number, information stored on a computer is often readable. We use strings to sort data, so that we can read it, and so that we can write programs to read the data and process it. Here are parts of three data files, where the data are stored as strings. It's important that the data is readable by you, not just by the computer. Although we could write programs even if everything was only a number, it will be easier to write programs to find patterns, knowledge and information and data, when the data is stored as strings.

The first part of a file is genomic data, stored in what's called fasta format. You write programs in this lesson to find proteins, and genes, and genomic data. The second part of a file is from a web-page. You will write programs to find links and other information in a web-page. Doing at a small scale what search engines like Google do, in ranking pages to be found by those doing web searches. The third part of a file is data from a CSV file, or Crime in Sacramento California in 2008. The CSV file, is a file in a special format. The CSV means, Comma Separated Values. You will write code in a later lesson to process CSV files.

A woman with curly hair and glasses, wearing a purple long-sleeved shirt, stands to the right of a presentation slide. The slide has a light purple background and contains the following text:

## Goals for Lesson

- Learn about the Java String class
  - Details, idioms, programming experience
  - Common String methods
- Learn about Java types and operators
  - Numeric types and operators as well as String
  - int and double types
- Finding patterns and information
  - Searching, repeating, reasoning

Duke UNIVERSITY

We have several goals for you to learn, as part of this lesson. You'll learn about the Java string class. You'll learn many details how to write programs with the string class, and how that's most often done. You'll learn common string functions, and how to read documentation to find out more about strings. You'll learn about Java types and operators. Here you'll learn more about Java's numeric types and operators, on those types. Which for you will be int or integer, and double or a floating point number. You'll learn about programming to find patterns in data, by searching for specific parts of a string. You'll repeat searches to find information and patterns, like all the links on a web page, or all the genes in a strand of DNA. Let's get started solving problems. Thank you.

### Video 3. Developing an Algorithm

Hello and welcome back. You are going to be working with strings that represent DNA, solving problems like searching for genes in them. This problem is a great one to learn on because even though we're going to start with a greatly simplified version of it, it's an important problem with real-world applications. Of course, the lessons you will learn about working with strings and programming in general will help you far beyond this problem domain. Whatever sorts of problems you want to solve, strings are likely to come up in them in one way or another, HTML, email or really anything that is text that is represented as a string. You're also going to learn some other important lessons as you work on these problems like how to do math in Java. Of course, math is also ubiquitous in programming sense, everything is a number. Perhaps most importantly, you will get more practice developing and implementing algorithms with the seven steps.



## DNA Concepts

**Nucleotide**

**Codon**

**Start Codon**

**Gene**

**Stop Codon**

```

ATCGTATACTGAAAACAGCTTTGAGATTGTTAAACACCGAAGATTATGATT
TCAGCGTTGAGGTCTAACCCTCAGCGATTATAGATGTGGAGGGTCTCTTCGC
TGTAAGCATACGACGGTCTAGAGCTGGGATGAGGCCGGAACACTGTTATGGGT
ACGGTTGTAAGTCTTGAACGCTCTGGAGGGTGGGCCGCCAAGTACTTGTCC
CAGGCGCGGGTACCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGTCC
GAAAGCCCTGGATTTCATCTTAGCATGCGGGAATCCGAAGTTGGAAGGTGAG
GACAGGAAACAATCTGATATGACCTGTAGATCAACTCTGAACCCGACAT
CCGAGCGATACCGACTCTACACGGGTGATGCATATCGTTGCGCTCTCTTTA
GAGATGATGCTGAATGGAAGAAAACCGCCACCCATCTCTAAGCGAACAGAA
AATAATGGAACCGCCGAACTATTTTCATAGAATGCAACGACGTTTGACA
ATGGCGTTCTATCCACTCAAATCTCCGTATACTAGCGTTATCACAGTC
ATTAAACGCCAAAAACAACGTATATGGCGTTGTAACGCTGCACATT
ACATCGTACAGTGCATCTTCTCCGGGAACCAAGCACATGACTACT
TACCAGGGAACGCAGATGTCTATCAGCACACCCGTTTGTATTGAGAG
TAAATGTACGCAATTTGAGTAATACACCCCTTCATGGTAGGGGACATGC
TACTGCAACCCTAGTATCACCTTAGAACGGGTACACACATTCGCAC
TACGCGGCAACTTGTTCGACGTTCTTGAGACGCTGTCGAGTGTTC
CTGGTCGGGACAATTATGACAAACGGCAGTCCAGCATCATATG
ACATTGGCTCCGTGTCACGCGCATTTGCTAGATCCGGGCA

```



Duke UNIVERSITY

Before we dive into DNA related problems, we need to give you a bit of domain knowledge, some terms and concepts related to working with DNA. Here is a string that could represent some piece of DNA. You will see that it's made up of four letters A T C and G. Each of these represents one nucleotide which are the basic building blocks of DNA. Three nucleotides together make a codon which each describe one amino acid. The **ATG codon** shown here is special in that it indicates the start of a gene. Accordingly, it's called the start codon and the **TAA codon** is also special in that it indicates the end of a gene, so it's called the stop codon. There are a couple other stop codons but for now, we're only going to think about TAA. Everything between and including these two codons makes up one gene.


## First Problem: Find a Gene (simplified)

- Find a "gene"
  - All text between "ATG" and "TAA"
- Greatly simplified to start
  - Real genes: all codons (multiple of 3 length)
  - Other stop codons than TAA

```

ATCGTATACTGAAAACAGCTTTGAGATTGTTAAACACCGAAGATTATGATT
TCAGCGTTGAGGTCTAACCCTCAGCGATTATAGATGTGGAGGGTCTCTTCGC
TGTAAGCATACGACGGTCTAGAGCTGGGATGAGGCCGGAACACTGTTATGGGT
ACGGTTGTAAGTCTTGAACGCTCTGGAGGGTGGGCCGCCAAGTACTTGTCC
CAGGCGCGGGTACCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGTCC
GAAAGCCCTGGATTTCATCTTAGCATGCGGGAATCCGAAGTTGGAAGGTGAGG

```



Duke UNIVERSITY

The first problem you're going to work on is finding a gene in a string which represents DNA. That is, you want to write a program which takes a string like this one and gives you all the text between it and including the start codon ATG and the stop codon TAA. You're going to start with a greatly simplified version of this problem, just finding those letters and all the text between them. You will not worry about the fact that real genes must be multiples of three in length because they're made up of codons or

that there are some other stop codons or a few other complexities to start with. As you master more string and algorithm concepts, you'll add features to your program making it more realistic with each step.

## Step 1: Work an Example Yourself


```

ATCGTATACTGAAAACAGCTTTGAGATTGTTAAACACCGAAGATTATGATT
TCAGCGTTGAGGTCTAACTACTCAGCGATTATAGATGTGGAGGGTCTCTCGC
TGTAAGCATACGACGGTCTAGAGCTGGGATGAGGCCCGAACACTGTTATGGGT
ACGGTTGTAAGTCTTGAAACGTCTTGGAGGGTGGGCCGCCCAAGTACTTGTCC
CAGGCGCGGGGTACCCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGTCC
GAAAGCCCTGGATTTCATCTTAGCATGCGGGAAATCCGAAGTTGGAAGGTGAGG

```

- Start with this string, find a gene

Answer: "ATGATTTTCAGCGTTGAGGTCTAA"



As always, the first thing you want to do is work on an instance of the problem yourself. Let's take this DNA sequence and find the first gene in it. Let's find the start codon, there it is. Now, let's start looking after it to find the stop codon and we find it right here. That means that we want to take all this text representing the nucleotides in this region as our answer, the gene that we found.


## Step 2: Write Down What You Just Did

```

ATCGTATACTGAAAACAGCTTTGAGATTGTTAAACACCGAAGATTATGATT
TCAGCGTTGAGGTCTAACTACTCAGCGATTATAGATGTGGAGGGTCTCTCGC
TGTAAGCATACGACGGTCTAGAGCTGGGATGAGGCCCGAACACTGTTATGGGT
ACGGTTGTAAGTCTTGAAACGTCTTGGAGGGTGGGCCGCCCAAGTACTTGTCC
CAGGCGCGGGGTACCCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGTCC
GAAAGCCCTGGATTTCATCTTAGCATGCGGGAAATCCGAAGTTGGAAGGTGAGG

```

- Start with this string, find a gene
  - I found the first occurrence of "ATG"
  - I started looking after the "ATG" for "TAA"
  - I took all the letters between (and including them) as my answer: "ATGATTTTCAGCGTTGAGGTCTAA"



Now that we have worked an example, we should write down what we just did. First, I found the first occurrence of ATG, then I started looking after the ATG for TAA. Last, I took all the letters between and including them as my answer ATG, ATT, TTC et cetera, all the way to TAA.

### Step 3: Generalize Steps

```
ATCGTATACTGAAAACAGCTTTGAGATTGTTAAACACCGAAGAGTTATGATT  
TCAGCGTTGAGGTCTAACCCTCAGCGATTATAGATGTGGAGGGTCTCTTCGC  
TGTAAGCATACGACGGTCTAGAGCTGGGATGAGGCCGAACACTGTTATGGGT  
ACGGTTGTAAGTCTTGAAACGTCTTGAGGGTGGGCCGCCCAAGTACTGTCC  
CAGGCGCGGGGTACCGTATGCTTATCTTAAGGAGACGCGGTGAGAGTGGTCC  
GAAAGCCCTGGATTATCTTAGCATGCGGGAATCCGAAGTTGGAAGGTGAGG
```

- Start with this string, find a gene
  - ① I found the first occurrence of "ATG"  
Always look for ATG Always look for TAA
  - ② I started looking after the "ATG" for "TAA"  
Always take letters between them
  - ③ I took all the letters between (and including them)  
as my answer: "ATGATTTTCAGCGTTGAGGTCTAA"  
Won't always be this string though



Okay, now that we wrote down what we did for that specific problem, we want to generalize it. Why did we look for ATG? We always want to look for it, that's the start codon. What about TAA after ATG? We always want to do that too, that is the stop codon and taking all the letters between and including them, we always want to do that too. The only thing that isn't really general here is the specific string we wrote down as our answer which was more of a descriptive note to ourselves than anything else.

### Before Proceeding, Some New Concepts

- How do we find ATG in a String?
  - How do we represent its position?
- How do we get all the letters in a range?



Now that we have a general algorithm, we'd like to turn it into code, but we need to learn some new Java concepts first. How do we find ATG in a string? For that matter, how do we even represent or talk about the position of something in a string and how would we get all the letters in a particular range in a string? You'll learn about these concepts then you'll be ready to turn this algorithm into code. Thank you.

## Video 4. Positions in Strings

To move forward implementing a gene finding program, we will explore a couple of important String topics in this video.

### Before Proceeding, Some New Concepts

- How do we find ATG in a String?
  - How do we represent its position? **Everything is a Number**

### Positions in a String: "Indexes"

dukeprogramming  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

- Number each position
  - Starts at 0
  - Called the "index" of that character
    - "The letter at index 3 is e"

The first of them is how we can represent the position of something in a String. To answer this question we return to the recurring concept that everything is a number. That is, we will give each position in the String a numeric index. Notice that these numbers start at 0 in the first position, not 1. This may seem a bit odd since we usually start counting from one, not zero. However, many programming languages start numbering sequences of things, such as Strings, from zero because it makes some tasks easier as we shall see later. These numbers which describe the positions in the String are called indices or indexes. Either word is okay as an okay plural of index. For example, if I wanted to talk about this e, I might say, the letter at index 3 is e. We have now answered the first question. We can represent the position with a number, and we can talk about the index in the String where we find the ATG.



## Before Proceeding, Some New Concepts

- How do we find ATG in a String?
  - How do we represent its position?
- How do we get all the letters in a range?

We can use a built-in method in the String class



## Built-in Method: substring

```
String s = "dukeprogramming";
```

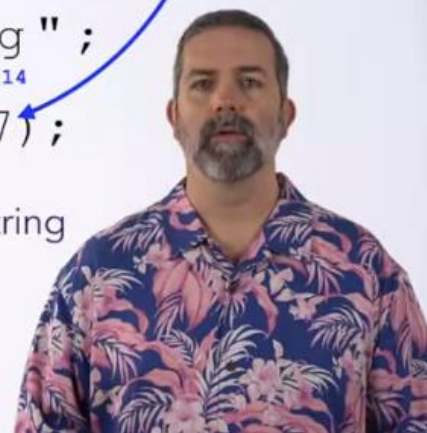
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

```
String x = s.substring(4, 7);
```

Start (included)

End (excluded)

- First: let's make a Java variable for our String
  - Put String literal in quotes
  - (otherwise, Java thinks it is a variable)
- Now, let's use the `substring` method



Now it is time to answer the second question. How could we get all the letters in a particular range, which we could now be a bit more precise, and say between two particular indices? One option would be for you write your own algorithm to do this. However, you can also use a built-in method of the String class. Whenever there is a built-in method to accomplish a particular task, it is better to use it than write your own. Not only does it save you work but the built-in method has already been heavily tested by expert programmers. So, you can be very confident that it works correctly.

For this particular task, you want to use the built-in **substring** method. However, before we show you how to do that, let's take this example String and make it an actual Java string assigned to a variable. Here we have a variable declared with the name `s` of type String and an equal sign to make an assignment statement. And a semicolon at the end of the line to end the statement. However, this isn't quite correct yet. We also need to put the String literal in **quotation marks** as shown here. If we did not put these and just wrote the word. Java would think `dukeprogramming` was the name of a variable and give us an error that it is undefined. By putting the text in quotation marks, Java knows that we want a String with that literal text. So now we have a valid statement which makes the variable `s` be the String `dukeprogramming`.

Next, you can see an example of using the **substring** method. Here, we declare another variable of type String called `x`, and assign it to the result of `s.substring(4,7)`. What do these numbers mean, and what


does this method call do? The first number specifies the index in `s` from which we want to start making our substring. The letter at this index will be included in the resulting String. The second number specifies the index in `s` where we want to stop making our substring. The letter in this index will be excluded from the resulting String. The method will stop right before it gets to that letter. This may seem odd, why would you want to specify the index after where you want to stop? There are a variety of reasons why this method and many others are designed this way. But one nice reason is that the length of the resulting String will be the difference between the two numbers. 7 minus 4 is 3, so we will get a three-letter String as our answer. In particular, you will end up with this three-letter String, made up of the letters from the indices 4, 5, and 6 of `s`. So, `x` will be the String `pro`.

## Other Useful Built-In Methods

```
String s = "dukeprogramming";  
           0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

method	value
<code>s.substring(4,7)</code>	"pro"
<code>s.length()</code>	15
<code>s.indexOf("program")</code>	4
<code>s.indexOf("g")</code>	7
<code>s.indexOf("f")</code>	-1
<code>s.indexOf("g",8)</code>	14
<code>s.startsWith("duke")</code>	true
<code>s.endsWith("king")</code>	false

`.endsWith` checks if the String ends with some other String (returns true or false)



Duke University

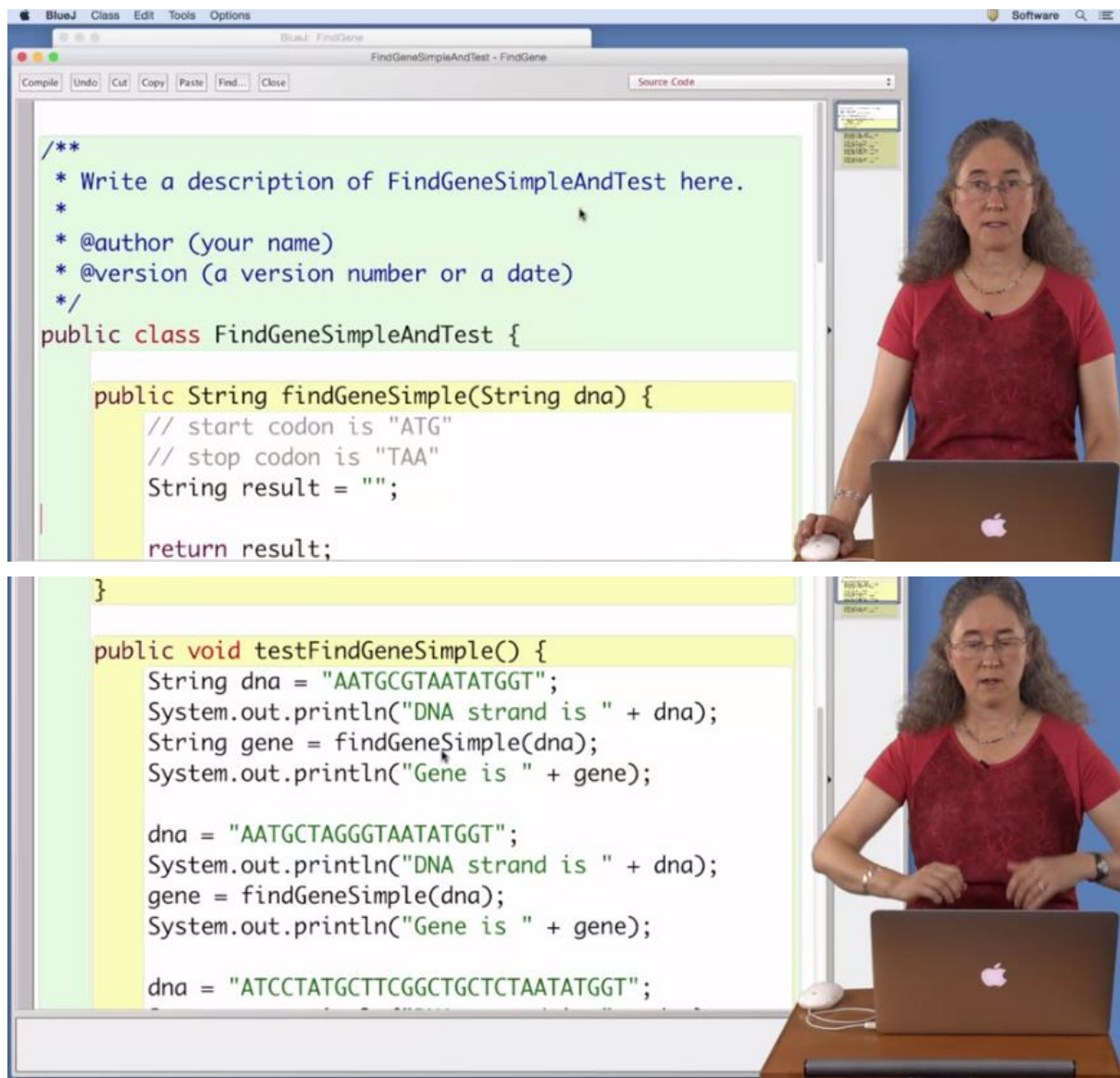
While you are learning about this built in method of String, let's take a moment to talk about a few other useful methods, and what they do. You just saw `substring` and learned how it gives you the letters in a particular range of indices. Another useful method is `.length`, which tells you how many characters are in the String. The String `s` has length 15. Notice that for a String of length 15 the valid indices are 0 through 14. If you try to access an index outside of this range your program will have an error with a String out of bounds exception. Another really useful method is `.indexOf`. You pass this method another String and it tries to find the first occurrence of that String within the String you called the method on. For example, here we have asked the `indexOf` method to find the String `program` within the String `s`. You can see that the first occurrence of `program` is right here, since it starts at index 4. That is the value that the method call would return. Here's another call to `indexOf`, `s.indexOf("g")`. Can you figure out what this would return? This will result in 7 since the first occurrence of the String `g` starts at index 7.

If you call `.indexOf` on a String that is not found, such as `.indexOf("f")`, it returns `-1`. You can also give `.indexOf` a second parameter specifying the index to start searching from. Here we have passed `8` as a second parameter. So the `.indexOf` method will ignore characters in the indices `0` to `7`, as it searches for `g`. It will then find this `g`, which is the first one when you start looking from index `8`, so then this method call evaluates to `14`. Another useful method is `.startsWith`, which tells you if a String starts with another String. Here you can see that `s` starts with `duke`, so this method call evaluates to `true`. Likewise, `.endsWith` checks to see if a String ends with another String. `s` does not end with `king`, so this method call evaluates to `false`.

Wow, that was a lot of methods, and a bunch of information. How would you know all of this without us telling you about every method in the String class? And should you be memorizing all of these details? Of course not! Programming is not about memorization. Although, as you program a lot, methods that you use commonly will naturally become familiar. Instead, you should learn how to make use of the language documentation, which describes all of the built-in classes and their methods. If you search in the internet for `java String`, your first result will probably be something from `docs.oracle.com`. Oracle is the company that makes Java and `docs.oracle.com` is the website where they host the language documentation. If you click on this link you will end up with a page which tell you all about the String class. If you scroll down a bit, you will find a rather long list of all the built-in methods in String. Here are a few of them, including a few that you have just learned about, `.indexOf` and `.length`. These entries give a brief description of the methods. And if you click on one of the method names, you will get a more detailed description of what that method does. There is also a documentation page on the course site at `dukelearntoprogram.com`. That page has simplified documentation of some important methods for quick reference. Great! Now you not only know about indices and Strings and some useful built in methods but also how to learn about other methods when you need them.

## Video 5. Translating into Code

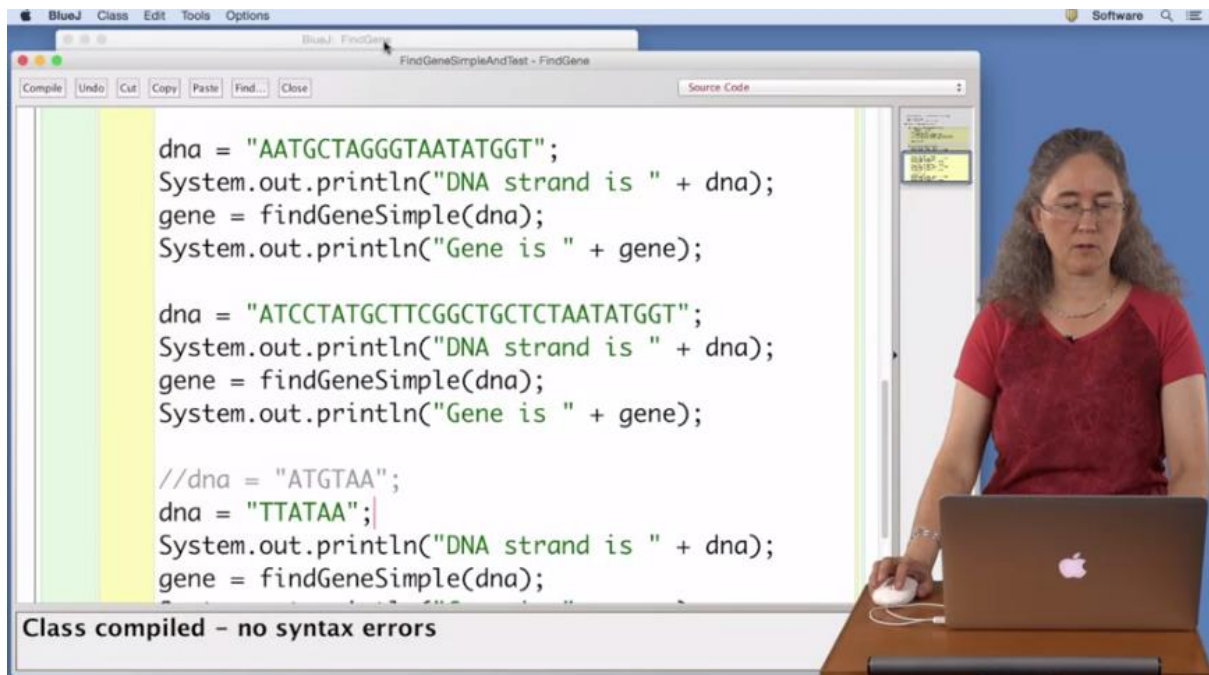
Hi, we're going to find a gene in a DNA strand. We're going to do a very simple algorithm in order to find our gene.



So I've started some code here. We have an algorithm method, called findGeneSimple. And we haven't done much with it yet, but what I've also done down here is I've written a couple of DNA strands that we will use to test the code that we write. So you can see here we have String dna. We're going to print our DNA stand. We're going to call findGeneSimple and then print out our gene. And then I just have done this with four different strands of DNA. So let's write the code now. So I've started by just saying, the resulting gene is just going to be called a variable called result. And I've set it to null, the null string. And if you remember from the video, we are going to look for the start codon in the strand of DNA.



And the start codon is ATG, the string ATG. So we can use the new string functions that we learned about. So I'm going to start by creating a variable that will hold the index position of the start codon. So we'll call it `startIndex`, and then we are going to look in the strand of DNA and we'll use the `indexOf` function to look for ATG. And the `indexOf` function is going to go through the string and it's going to stop when it finds ATG and it's going to return the index location of where it starts. So that'll be the starting position of our gene. We also have to look for the stop codon, and that is TAA. So we're going to create a variable for that called `stopIndex`, for the position of where the stop codon is. Again we will look in our DNA strand, DNA and we'll use `indexOf`. But if you just look for TAA, it will start at the beginning of the string. And we want to look for the stop codon after the start codon. So we know where the start codon is. It's in the variable `startIndex`. And we want to start looking past that. So you can add a second parameter to the `indexOf` function, and so we will add that and say, start looking where ATG is, which is `startIndex + 3`, which is the length of ATG. So now we have the `startIndex` and the `stopIndex`, `startIndex` of ATG and the `stopIndex` of TAA. And now we want the strand that includes those two and everything in between it. So I'm going to call that our result. Again we'll use a string function. So we'll use the string function called `substring`. And the way `substring` works is you have to say I want a piece of a string and I want to start, where do we want to start? We want to start where ATG is, so we'll start at the `startIndex` and then the rest of our gene is going to be everything past ATG up until TAA, which is at the `stopIndex`. But we also want to include TAA so we'll add 3. So that means take our `substring` from where ATG starts, go all the way to the `stopIndex + 3`, which is the first character after TAA. And the way `substring` works is it says start at the first `startIndex` and then go all the way up to, but not including, the second parameter. And then let's compile this and see if it works. Okay, so it compiles fine. So let's go over here, and we'll create a new `findGene`. And now we'll run our test method that we wrote. And there it is. So we can see, here's the first DNA strand. And you look for ATG, there it is. And then look for TAA and there's the first TAA that's after it. And you can see there's the gene that we found, right here. And then the next example here's ATG. We look for TAA all the way here. And you can see here that's the gene we found. Here's the third example, ATG all the way to TAA. You can see that we've got a big gene there. And then here's the last example. ATGTAA so that works good too.



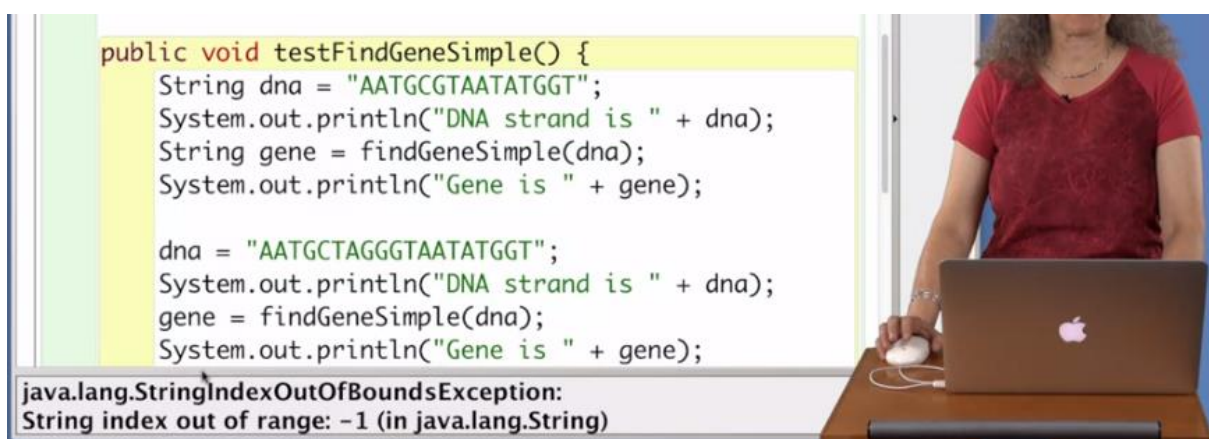
```
BlueJ Class Edit Tools Options
FindGeneSimpleAndTest - FindGene
Compile Undo Cut Copy Paste Find... Close Source Code

dna = "AATGCTAGGGTAATATGGT";
System.out.println("DNA strand is " + dna);
gene = findGeneSimple(dna);
System.out.println("Gene is " + gene);

dna = "ATCCTATGCTTCGGCTGCTCTAATATGGT";
System.out.println("DNA strand is " + dna);
gene = findGeneSimple(dna);
System.out.println("Gene is " + gene);

//dna = "ATGTAA";
dna = "TTATAA";
System.out.println("DNA strand is " + dna);
gene = findGeneSimple(dna);

Class compiled - no syntax errors
```

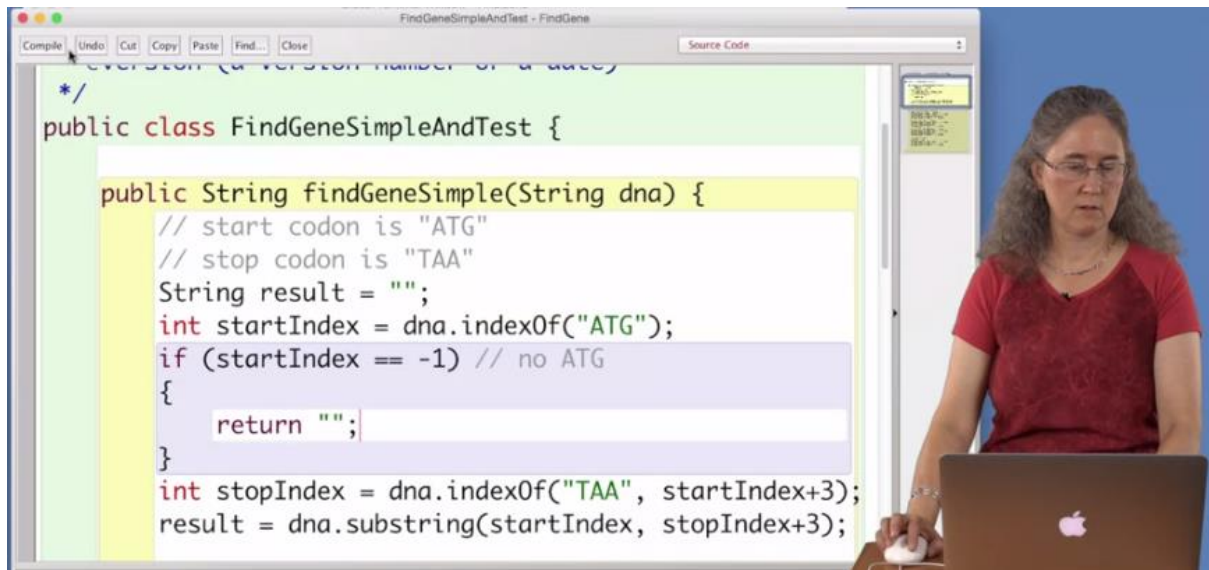


```
public void testFindGeneSimple() {
    String dna = "AATGCGTAATATGGT";
    System.out.println("DNA strand is " + dna);
    String gene = findGeneSimple(dna);
    System.out.println("Gene is " + gene);

    dna = "AATGCTAGGGTAATATGGT";
    System.out.println("DNA strand is " + dna);
    gene = findGeneSimple(dna);
    System.out.println("Gene is " + gene);
}

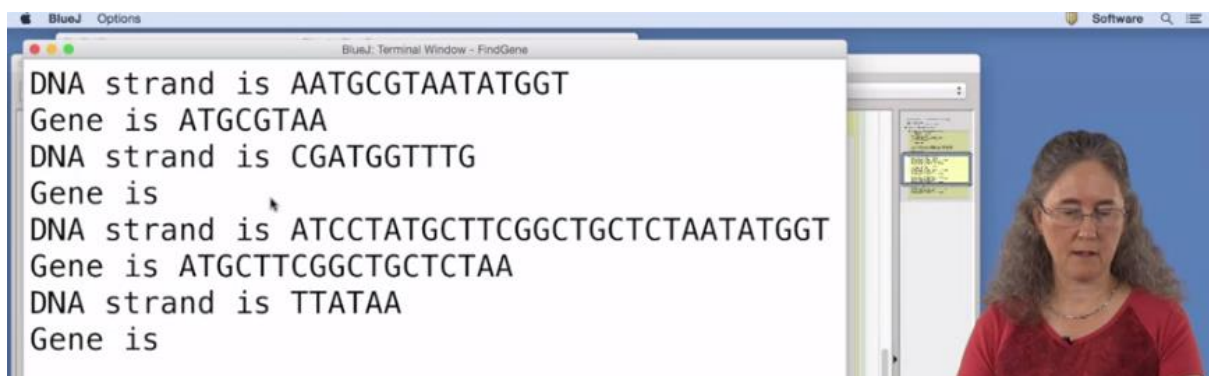
java.lang.StringIndexOutOfBoundsException:
String index out of range: -1 (in java.lang.String)
```

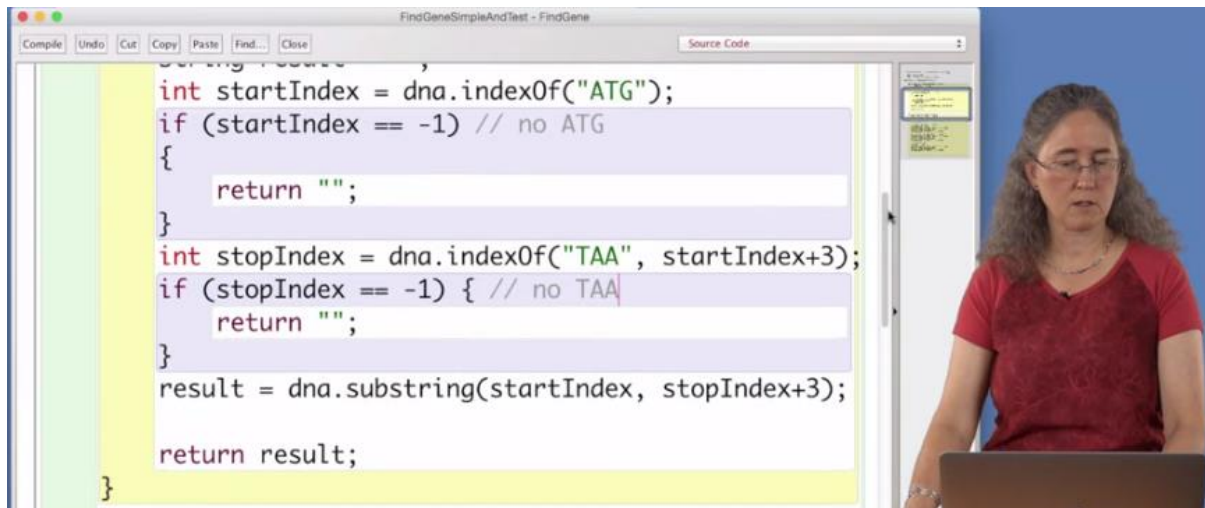
But what happens if ATG is not there? Or what happens if TAA is not there? So let's look at our example here, and let's change one of our strings. Let's change this string. We'll just save this one. And let's change it to, A string that does not have ATG in it. Let's see what happens if we run that example. So we're going to come back over here. And we'll run our testFind, and we got an error down here. You can see, it says `StringIndexOutOfBoundsException, -1`. String index out of range. What happened? Let's go look at our code. So we looked for ATG and we didn't find it. And what the `indexOf` function does if it doesn't find the string you are looking for, it returns -1. So `startIndex` has the value -1 and then we are trying to build a string starting at -1 and -1 doesn't exist. So we got an error. So what are we going to do? We need to fix this code. So let's check right after we ask for ATG, **let's put an if statement in there to check to make sure that ATG was there.**



So we can ask if startIndex is equal to -1, that means there is no ATG. In that case, there is no gene. There's no gene if there's no ATG, there's no start. So what we'll do is we'll just return the empty string. So if you compile this now and if we run it. So we'll create findGene. And then we'll run testFind, and it ran, but there's no gene here. That's because there's no ATG. Let's try another example. We'll change our code again to give another example. Let's just change this one here.

Let's create a string where we have ATG but we don't have TAA. So we have the start codon, but not the stop codon. So I'll create a string, CGATGGTTTAAAAGT. So there's no, there is, let's get rid of it. G, there we go. So now we have ATG right here and to the right of it there is no TAA. So let's compile and we'll run it. And you can see it's not there.





So right here we can check and see if stop. If the stopIndex equals -1, at that point we know there's no stop codon and so there's no gene. And so we can again, just return the empty string. Return, Empty string. So that's just a safe way, if there's no start codon, or no stop codon. Let's add a comment here. This is a case for no TAA. We'll compile our code and we'll just double check and make sure that works. So it compiles and we'll run it. And now we have, our first string is good, we find a gene. Our second string, there is no gene. We have an ATG but we don't have a TAA. This string looks good. The string has no ATG. It actually does have a TAA but that doesn't matter because it has no ATG. And so there is no gene for that one either. So anyway that is the way to find a simple gene and a strand of DNA. We look for the start codon and then if there's a start codon, we look past it for the stop codon. And if we find both of those, then we can return the start codon, the stop codon, and everything in between it as the gene. Thank you.

## Video 6. Java Math

All right, now you have an implementation of the rather oversimplified version of this program. It just looks for ATG and then TAA, and gives back everything in between. But real genes have to be a multiple of 3 in length, since they're made of codons. Each of which is 3 nucleotides long.



## Previous Algorithm: Gross Oversimplification

- Just finds ATG....TAA
  - Real genes: made of codons (3 nucleotides)

AGATGCGATACGCTTAATC  
AGATGCGATACGCTAATC

- Let's fix this:
  - New algorithm: better, but still a simplification

For example, this string is a valid gene. You can see here how it can be divided into codons starting with ATG and ending with TAA. However this string is not valid. Even though it has ATG and it has TAA when we look between them we don't find a valid sequence of codons. Now, let's make our algorithm a little bit more realistic. You will fix this aspect of your algorithm, it will still be a simplification, but a bit more realistic. At this point, it's really good to note that starting with a simple version and adding features, is not only a useful technique, for you, as you learn new concepts. It let's us introduce a few concepts at a time. But also important when writing real, large complex problems.

## How To Tell In General?

- Saw two examples: one OK, one not.

AGATGCGATACGCTTAATC  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

AGATGCGATACGCTAATC  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

- How to tell in general?


Okay, so you just saw two examples of DNA strings. One that has a valid gene and one that does not. How can you tell, in general? What is the change you need to make to your algorithm? It might be useful to show the indices as we have shown here and highlight the location of the start and stop codons.

Do you see how to algorithmically tell the difference? If not, that's fine. It can be hard to spot patterns. But you will get better at it with practice.

## More Examples?

String	Index of ATG	Index of TAA	Valid?
AGATGCGATACGCTTAATC	2	14	Yes
AGATGCGATACGCTAATC	2	13	No
GATGCGATACGCTTAATC	1	13	Yes
ATGCGATACGCTAATC	0	11	No
ATGTTAAACAT	0	4	No
ATGTACTAAACAT	0	6	Yes

Pattern may still be hard to see...



One great technique to find patterns is to make a table. You might remember that we did this in some of our examples. We can then add more examples to our table as I'm showing here to help us see the pattern. Some with yes answers and some with no answers. Maybe you see the pattern now or maybe it is still hard to see. What can you do for pattern still isn't clear? Maybe adding more rows will help or maybe not.

## More Examples?


String	Index of ATG	Index of TAA	Difference	Valid?
AGATGCGATACGCTTAATC	2	14	12	Yes
AGATGCGATACGCTAATC	2	13	11	No
GATGCGATACGCTTAATC	1	13	12	Yes
ATGCGATACGCTAATC	0	11	11	No
ATGTTAAACAT	0	4	4	No
ATGTACTAAACAT	0	6	6	Yes

Instead we might start exploring the relationships between the items in the table. Here we have added another column for the difference in the index of the stop codon and the index of the start codon. The ones that have yes answers have differences of 6 and 12. And the ones that have no answers have

differences of 4 and 11. What do 12 and 6 have in common that 11 and 4 do not? We might be able to think of a lot of things, 6 and 12 are both multiples of 6. But that doesn't make sense in the context of this problem. If we did more examples, we could find ones with yes answers that have 3 or 9 or 15. However, 6 and 12 as well as 3, 9 and 15, are all multiples of 3. This relationship does make sense. As we know that the length must be a multiple of 3.

## Now We Need Some Math!

- Need to do some math in Java
  - May recall "mod" from Course 1
    - $x \% y$ : remainder when  $x$  is divided by  $y$
  - Can use other mathematical operators:
    - $+$ ,  $-$ ,  $*$ ,  $/$
    - $==$ ,  $!=$ ,  $<$ ,  $<=$ ,  $>$ ,  $>=$
  - Can combine simple expressions into more complex ones:
    - $(a - b) \% 3 == 0$



**Duke**  
UNIVERSITY

Now that you know the relationship, you know that you want to look for. You need to do some math in Java. You need a way to ask if the difference between two numbers is a multiple of three. If you took course one with us you might remember the mod operator which gives you the remainder when you do division.  $x \bmod y$  written  $x \% y$  means divide by, divide  $x$  by  $y$ , but give me the remainder, not the quotient. This can help you with the problem at hand since a number that is a multiple of three has a remainder of zero when divided by three. That is, if  $x$  minus three is equal to zero, then  $x$  is a multiple of three. You can use other mathematical operators in Java,  $+$  minus, times, divide, they're all valid. You can also use equals equals to see if two numbers are the same. Not equals to see if they're different and less than, less than, or equals, greater than, or greater than or equals to check for inequalities. You can also combine simpler expressions into more complex expressions. This expression checks if  $(a - b) \bmod 3$  is equal to 0. It is evaluated by first evaluating  $(a - b)$ , then taking that result and doing mod 3 on it. Then finally taking that result and checking if it's equal to 0. This is pretty much exactly what you need for the prominent hand. To see if the difference between two things is a multiple of three.

## Arithmetic with Java

- Two different types of numbers
  - `int` represents integers ... -2, -1, 0, 1, 2, ...
  - `double` represents real numbers, in programming "floating point" 1.2, 3.457, ...



While you are learning about math in Java, there are several different types of numbers, actually there are some variations on these. But, you don't need to worry about that right now. One type of number is `int`, which represents integer numbers like -2, -1, 0, 1, 2, etc. Ints can't have a fractional part. The other type is `double`, which represents real numbers. Ones with fractional parts like 1.2 or 3.4, 5.7, of course you could also represent 3 which is 3.0 with a double. You only want to use doubles when you need to since they have some behaviors that can be really confusing to novice programmers.

- Arithmetic with integers
  - Always yields an integer
    - $5/2$  ? 2
    - $100 / 3 * 4$  ? 132
    - $100 * 4 / 3$  ? 133

Duke  
UNIVERSITY



One word of caution about integers however is that **math on integers always yields integers**. So what do you get if you divide 5 by 2? If you are thinking 2.5, remember that you can only get an integer result. So you get 2. What about 100 divided by 3 times 4? You will get 132 since 100 divided by 3 is 33 and 33 times 4 is 132. So what if you do 100 times 4 divided by 3? Seems like you should get the same answer right? Well actually now you get 133, why? 100 times 4 is 400 and 400 divided by 3 is a 133. These sorts of issues should not come up in these courses, but are good to be wary of if you're doing integer division.



## Order Of Operations

- Java has Order Of Operations (like math)
  - $a + b * c$  means  $a + (b * c)$
  - $a - b \% 3$  means  $a - (b \% 3)$
  - $a + b == c - d$  means  $(a + b) == (c - d)$
- Generally make sense, like math
  - Not sure? Use parentheses



Another thing to know about math in Java is that it has order of operations. Rules like math in programmer speak these rules are called precedents and associativity. As with math,  $a + b \text{ times } c$  means to do  $b \text{ times } c$  first then add that result to  $a$ . What if you have mod? It has the same precedence as division, meaning it takes place at the same place in the order of operations. So  $a - b \text{ mod } 3$  means to  $(b \text{ mod } 3)$ , then subtract that result from  $a$ . This is why we put parentheses around  $a - b$  earlier when we wanted to do the minus first, then take its result mod 3. Comparisons for equality happened very late in the order of operations.  $A \text{ plus } B \text{ equals } C \text{ minus } D$  means to first do  $A \text{ plus } B$  then do  $C \text{ minus } D$  and finally compare the two results to see if they are equal to each other. These rules are typically like they are in math. Multiplication and division comes first. Then addition and subtraction. Also like math you can use parentheses to group things so that they come first. If you're not sure what order things will happen you can always use parentheses to be explicit and be sure you get what you want. Okay, now that you know about math in Java, it's time to go improve your gene finding algorithm. Thank you.

## Programming Exercise: Finding a Gene and Web Links

A PDF of the programming exercise can be found in the **Resources** tab.

For files related to this assignment, visit the DukeLearnToProgram Project Resources page for this course: <http://www.dukelearntoprogram.com/course2/files.php>. Also linked in the **Resources** tab.

You can also find the frequently asked questions page for this course's assignments on DukeLearnToProgram: <http://www.dukelearntoprogram.com/course2/faq.php>. Also linked in the **Resources** tab.

### Part 1: Finding a Gene - Using the Simplified Algorithm

This assignment is to write the code from the lesson from scratch by following the steps below. This will help you see if you really understood how to put the code together, and might identify a part that you did not fully understand. If you get stuck, then you can go back and watch the coding videos that go with this lesson again. We recommend you try this with many of the future Java coding examples before starting programming exercises.

Specifically, you should do the following:

1. Create a new Java project named StringsFirstAssignments. You can put all the classes for this programming exercise in this project.
2. Create a new Java Class named Part1. The following methods go in this class.
3. Write the method findSimpleGene that has one String parameter dna, representing a string of DNA. This method does the following:
  - Finds the index position of the start codon "ATG". If there is no "ATG", return the empty string.
  - Finds the index position of the first stop codon "TAA" appearing after the "ATG" that was found. If there is no such "TAA", return the empty string.
  - If the length of the substring between the "ATG" and "TAA" is a multiple of 3, then return the substring that starts with that "ATG" and ends with that "TAA".
4. Write the void method testSimpleGene that has no parameters. You should create five DNA strings. The strings should have specific test cases, such as: DNA with no "ATG", DNA with no "TAA", DNA with no "ATG" or "TAA", DNA with ATG, TAA and the substring between them is a multiple of 3 (a gene), and DNA with ATG, TAA and the substring between them is not a multiple of 3. For each DNA

string you should:

- Print the DNA string.
- See if there is a gene by calling `findSimpleGene` with this string as the parameter. If a gene exists following our algorithm above, then print the gene, otherwise print the empty string.

## Part 2: Finding a Gene - Using the Simplified Algorithm Reorganized

This assignment will determine if a DNA strand has a gene in it by using the simplified algorithm from the lesson, but organizing the code in a slightly different way. You will modify the method `findSimpleGene` to have three parameters, one for the DNA string, one for the start codon and one for the stop codon.

Specifically, you should do the following:

1. Create a new Java Class named `Part2` in the `StringsFirstAssignments` project.
2. Copy and paste the two methods `findSimpleGene` and `testSimpleGene` from the `Part1` class into the `Part2` class.
3. The method `findSimpleGene` has one parameter for the DNA string named `dna`. Modify `findSimpleGene` to add two additional parameters, one named `startCodon` for the start codon and one named `stopCodon` for the stop codon. What additional changes do you need to make for the program to compile? After making all changes, run your program to check that you get the same output as before.
4. Modify the `findSimpleGene` method to work with DNA strings that are either all uppercase letters such as `"ATGGGTTAAGTC"` or all lowercase letters such as `"gatgctataat"`. Calling `findSimpleGene` with `"ATGGGTTAAGTC"` should return the answer with uppercase letters, the gene `"ATGGGTTAA"`, and calling `findSimpleGene` with `"gatgctataat"` should return the answer with lowercase letters, the gene `"atgctataa"`. HINT: there are two string methods `toUpperCase()` and `toLowerCase()`. If `dna` is the string `"ATGTAA"` then `dna.toLowerCase()` results in the string `"atgtaa"`.

## Part 3: Problem Solving with Strings

This assignment will give you additional practice using String methods. You will write two methods to solve some problems using strings and a third method to test these two methods.

Specifically, you should do the following:

1. Create a new Java Class named `Part3` in the `StringsFirstAssignments` project. Put the following

methods in this class.

2. Write the method named `twoOccurrences` that has two `String` parameters named `stringa` and `stringb`. This method returns `true` if `stringa` appears at least twice in `stringb`, otherwise it returns `false`. For example, the call `twoOccurrences("by", "A story by Abby Long")` returns `true` as there are two occurrences of "by", the call `twoOccurrences("a", "banana")` returns `true` as there are three occurrences of "a" so "a" occurs at least twice, and the call `twoOccurrences("atg", "ctgtatgta")` returns `false` as there is only one occurrence of "atg".

3. Write the void method named `testing` that has no parameters. This method should call `twoOccurrences` on several pairs of strings and print the strings and the result of calling `twoOccurrences` (`true` or `false`) for each pair. Be sure to test examples that should result in `true` and examples that should result in `false`.

4. Write the method named `lastPart` that has two `String` parameters named `stringa` and `stringb`. This method finds the first occurrence of `stringa` in `stringb`, and returns the part of `stringb` that follows `stringa`. If `stringa` does not occur in `stringb`, then return `stringb`. For example, the call `lastPart("an", "banana")` returns the string "ana", the part of the string after the first "an". The call `lastPart("zoo", "forest")` returns the string "forest" since "zoo" does not appear in that word.

5. Add code to the method `testing` to call the method `lastPart` with several pairs of strings. For each call print the strings passed in and the result. For example, the output for the two calls above might be:

- The part of the string after an in banana is ana.
- The part of the string after zoo in forest is forest.

## Part 4: Finding Web Links

Write a program that reads the lines from the file at this URL location (<http://www.dukelearntoprogram.com/course2/data/manylinks.html>) and prints each URL on the page that is a link to `youtube.com`. Assume that a link to `youtube.com` has no spaces in it and would be in the format (where [stuff] represents characters that are not verbatim): "`http:[stuff]youtube.com[stuff]`"

Here are suggestions to get started.

1. Create a new Java Class named `Part4` in the `StringsFirstAssignments` project and put your code in that class.
2. Use `URLResource` to read the file at <http://www.dukelearntoprogram.com/course2/data/manylinks.html> word by word.
3. For each word, check to see if "`youtube.com`" is in it. If it is, find the double quote to the left and right of the occurrence of "`youtube.com`" to identify the beginning and end of the URL. Note, the double quotation mark is a special character in Java. To look for a double quote, look for (`\`),



since the backslash (\) character indicates we want the literal quotation marks (") and not the Java character. The string you search for would be written "\" for one double quotation mark.

4. In addition to the String method `indexOf(x, num)`, you might want to consider using the String method `lastIndexOf(s, num)` that can be used with two parameters `s` and `num`. The parameter `s` is the string or character to look for, and `num` is the last position in the string to look for it. This method returns the last match from the start of the string up to the `num` position, so it is a good option for finding the opening quotation mark of a string searching backward from "youtube.com." More information on String methods can be found in the Java documentation for Strings: <http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>.

Caution: The word Youtube could appear in different cases such as YouTube, youtube, or YOUTUBE. You can find the URLs more easily by converting the string to lowercase. However, you will need the original string (with uppercase and lowercase letters) to view the YouTube URL to answer a quiz question because YouTube links are case sensitive. The link [https://www.youtube.com/watch?v=ji5\\_MqicxSo](https://www.youtube.com/watch?v=ji5_MqicxSo) is different than the link [https://www.youtube.com/watch?v=ji5\\_mqicxso](https://www.youtube.com/watch?v=ji5_mqicxso), where all the letters are lowercase.