DNA Processor Package for Use with JAVA ^{TM} Programming Language

Version 1.0.0

Suphanat “Soup” Isarangkoon Na Ayutthaya

February 29, 2020

Contents

Part I

Nucleotide Class

Chapter 1

Introduction to Nucleotide Class

1.1 What Nucleotide Class Is

Nucleotide Class is a class that represents each nucleotide of the DNA Molecule. It consists of two parameters that define each object of type Nucleotide, the nucleotide type and whether that nucleotide is a dideoxynucleotide (also known as a chain terminator nucleotide). This class has many built-in methods that are useful for DNA sequence processing. The Strand object, like any other object must be instntiated (defined) in the *main* class[1] using construtor methods. These construtor methods can be found in Part I, Chapter 2.

1.2 What A Nucleotide Is in Biology

A DNA Nucleotide is a unit of sugar, a phosphate group, and a nitrogenous base that are strung together in a DNA Strand. The nitrogenous base in a nucleotide can be divided into two groups: purine and pyrimidine. Adenine (A) and Guanine (G) are Purines, which consist of two rings, while Cytosine (C) and Thymine (T) are Pyrimidines, which consits of one ring. Purines and Pyrimidines bond together in a double-helix strand as a secondary structure, where A bonds with T and C bonds with G. This is because A-T pair has two hydrogen bonds,

[1] A *main* class is a class that is defined as having the only method having the header of "public static void main(String[] args)" Every functioning Java Program needs one class with a *main* method as the *main* class serves as the entry point for the program and is the class name passed to the java interpreter command to run the application; therefore, the code in the *main* method executes first when the program starts and is the control point where the data can be worked on.[1]

while C-G pair has three hydrogen bonds, therefore, A will never pair with C and G will never bond with T, as the number of hydrogen bonds dictate that these "interlocking" must happen in this specific way.[2] Since A only binds with T, A is said to be a *"complementary"* base of T, and vice versa; in this same logic, since C always binds to G, C is said to be a complementary base of G and vice versa. This is because they are the opposite base of the other base, and therefore "complements" the other base. Figures of the general structures of a DNA nucleotide and dideoxynucleotide as well as all four DNA nucleotides can be seen below.

A dideoxynucleotide is a unit of sugar, a phosphate group, and a nitrogenous base like a DNA nucleotide but it lacks one hydroxy group (OH) at 3'. Dideoxynucleotide is used in Sanger's Sequencing Technique as a chain terminator during the process of new DNA synthesis[2], Dideoxynucleotide consists of the same nitrogenous bases as a DNA nucleotide with the same hydrogen bonding dictating what nitrogenous base can be bonded to what nitrogenous base. Please note that DNA nucleotide can also be bonded to dideoxynucleotide.

Figures of the general structures of a DNA nucleotide and dideoxynucleotide as well as all four DNA nucleotides can be seen below.

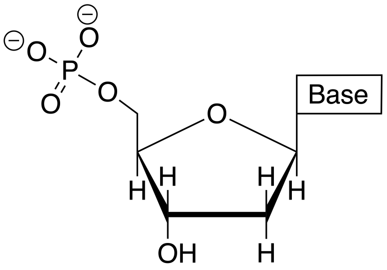

Figure 1: The molecule of a DNA nucleotide with any nitrogenous base (labelled as "Base")[2]

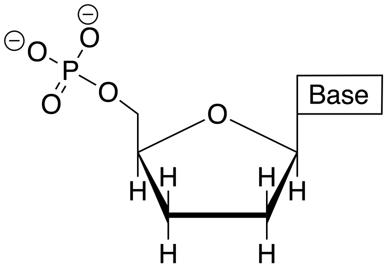

Figure 2: The molecule of a dideoxynucleotide with any nitrogenous base (labelled as "Base"). In this figure, the missing hydroxy (OH) group can be noted.[2]

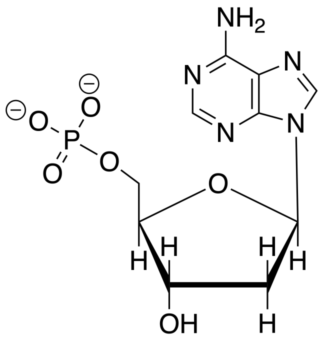

Figure 3: ’A’ (Adenosine) Nucleotide; Deoxyadenosine 5’-monophosphate) [2]

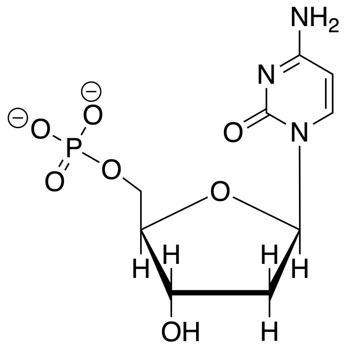

Figure 4: ’C’ (Cytosine) Nucleotide; Deoxycytidine 5’-monophosphate[2]

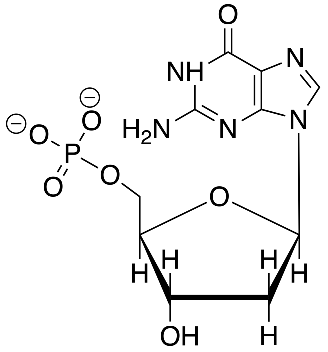

Figure 5: ’G’ (Guanine) Nucleotide; Deoxyguanosine 5’-monophosphate) [2]

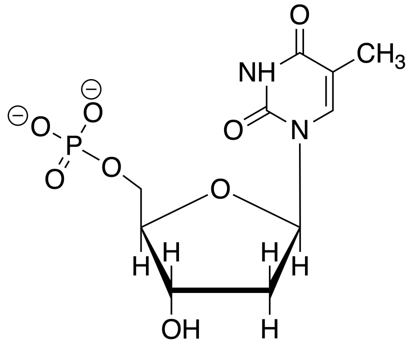

Figure 6: ’T’ (Thymine) Nucleotide; Deoxythymidine 5’-monophosphate[2]

Chapter 2

Constructor

A Construtor Method is a method that is called during the creation of an object (instatiation process) by the *main* class.[3] It performs a function to allow the for the new instant of the object "Nucleotide" to be created with its field defined.

2.1 Summary

Constructor and Description
Nucleotide () Create a Nucleotide without specifying what the nitrogenous base is
Nucleotide (*char* base) Create a Nucleotide with the base specified as a *char*
Nucleotide (*String* baseString) Create a Nucleotide with the base specified as a *String* object.

2.2 Constructor Detail

Nucleotide ()

Create a Nucleotide without specifying what the nitrogenous base is, thus the nucleotide is defined as a blank nucleotide (-)

Nucleotide (*char* base)

Create a Nucleotide with the base specified as a *char*. The *char* can be either ’*A*’, ’*C*’, ’*G*’, or ’*T*’. If any other *char* is inputted, the Nucleotide will be created as a generic Nucleotide with an unspecied nitrogenous base, ’*N*’.

Parameters:
base - A *char*

Nucleotide (*String* baseString)

Create a Nucleotide with the base specified as a *String* object. The *String* can be either “*a*”, “*A*”, “*c*”, “*C*”, “*g*”, “*G*”, “*t*”, or “*T*”. The input *String* is not case-sensitive, therfore, for example, “*a*” or “*A*” input will result in the same Nucleotide ’*A*’ being created. If any other *String* is inputted (including those that are longer than one-charcater long), the Nucleotide will be created as a generic Nucleotide with an unspecied nitrogenous base, ’*N*’.

Parameters:
base - A *String*

Chapter 3

Method

3.1 Summary

Modifier and Type	Method and Description
void	**set**(*char* **base**) Change a certain Nucleotide to another type of Nucleotide with a different nitrogenous base
void	**set**(*String* **base**) Change a certain Nucleotide to another type of Nucleotide with a different nitrogenous base
char	**getChar**() Returns the *char* value corresponding to the type of the current Nucleotide
boolean	**dideoxyOrNot**() Returns *true* if the current Nucleotide is a dideoxynucleotide (Not Currently Supported)
char	**giveComplementaryChar**() Returns a *char* value corresponding to the type of Nucleotide complementary to the current Nucleotide
void	**setComplementary**() Change the current Nucleotide into its complementary Nucleotide

static Nucleotide	**random**() returns a Nucleotide object of a completely random type
boolean	**equals**(Nucleotide **n**) returns *true* if current Nucleotide is the same type as the Nucleotide n

3.2 Method Detail

void **set**(*char* **base**)

Sets the current Nucleotide to the type with nitrogenous base corresponding to the *char* inputted as base; for example, if ’A’ is inputted as base, then the current Nucleotide will be set to a Nucleotide with an Adenosine nitrogenous base

Parameters:

base - A *char*

void **set**(*String* **base**)

Sets the current Nucleotide to the type with nitrogenous base corresponding to the *String* **base**; for example, if ”“A” is inputted as base, then the current Nucleotide will be set to a Nucleotide with an Adenosine nitrogenous base. The *String* inputted must be one-character long and must be either “A”, “a”, “C”, “c”, “G”, “g”, “T”, or “t”, otherwise, the current the Nucleotide will be set to a generic Nucleotide with an unspecied nitrogenous base, ’N’

Parameters:

base - A *String*

char **getChar**()

Returns the *char* corresponding to the current Nucleotide’s nitrogenous base. For example, if this Nucleotide has an Adenosine nitrogenous base, ’A’ will be returned when this method is called

boolean **dideoxyOrNot**()

Returns *true* if the current Nucleotide is a dideoxynucleotide. This feature is currently not supported, so whenever this method is called, it will always return *false*

char **giveComplementaryChar**()
Returns a *char* value corresponding to the type of Nucleotide complementary to the current Nucleotide. Complementary base is the base that is the opposite to the current Base – that is it is the base that would bind/pair up with the current base; Adenosine (A) is complementary to Thymine (T) and vice versa, and Cytosine (C) is complementary to Guanine (G) and vice versa. Therefore, for example, if the current Nucleotide has nitrogenous base 'A' (Adenosine), then when this method is called 'T' will be returned

void **setComplementary**()
Changes the current Nucleotide to its complementary Nucleotide. For example, if the current Nucleotide has nitrogenous base 'A' (Adenosine), then it will be set instead to have nitrogenous base 'T' (Thymine)

static Nucleotide **random**()
Returns a Nucleotide object with a completely random nitrogenous base

boolean **equals**(Nucleotide **n**)
Returns *true* if the current Nucleotide has the same nitrogenous base as the Nucleotide **n** that is inputted as a parameter; otherwise, returns *false* **Parameters:** **n** - A Nucleotide

Part II

Strand Class

Chapter 4

Introduction to Strand Class

4.1 What Strand Class Is

Strand Class is an ArrayList-based class that simply contains many Nucleotide objects. It is meant to represent a single DNA srand – a polynucleotide strand (not double-helix) made up of many nucleotides strung together in real life. This class has many built-in methods that are useful for DNA sequence processing. The Strand object, like any other object must be instntiated (defined) in the $main$ class using construtor methods. These construtor methods can be found in Part II, Chapter 5. it should be noted that a DNA strand defined by Strand class runs from 3' end to 5' end. (To know what 3' and 5' end is, please refer to Part II,Chapter 4, Section 4.2.)

4.2 Basic Biology of DNA Strands and DNA Double-Helix

DNA Strands are formed from many nucleotides strung together. To join each nucleotide together, the phosphate group bonded to the 5' carbon of one nucleotide is bonded to the 3' carbon of another nucleotide. These bonds are called "*phosphodiester linkages*", and are strong covalent bonds that link multiple nucleotides; the stringing of these many nucleotides constitutes a backbone of a polynucleotide strand. Any combination of nucleotide can be strung together in a DNA strand. DNA strands usually exist in the form of a secondary structure called a double-helix where two strands are bonded via hydrogen bonding to a complementary strand – another strand with all the bases complementary to this current one (see complementary base pairing in Chapter 1, Section 1.2). [2]

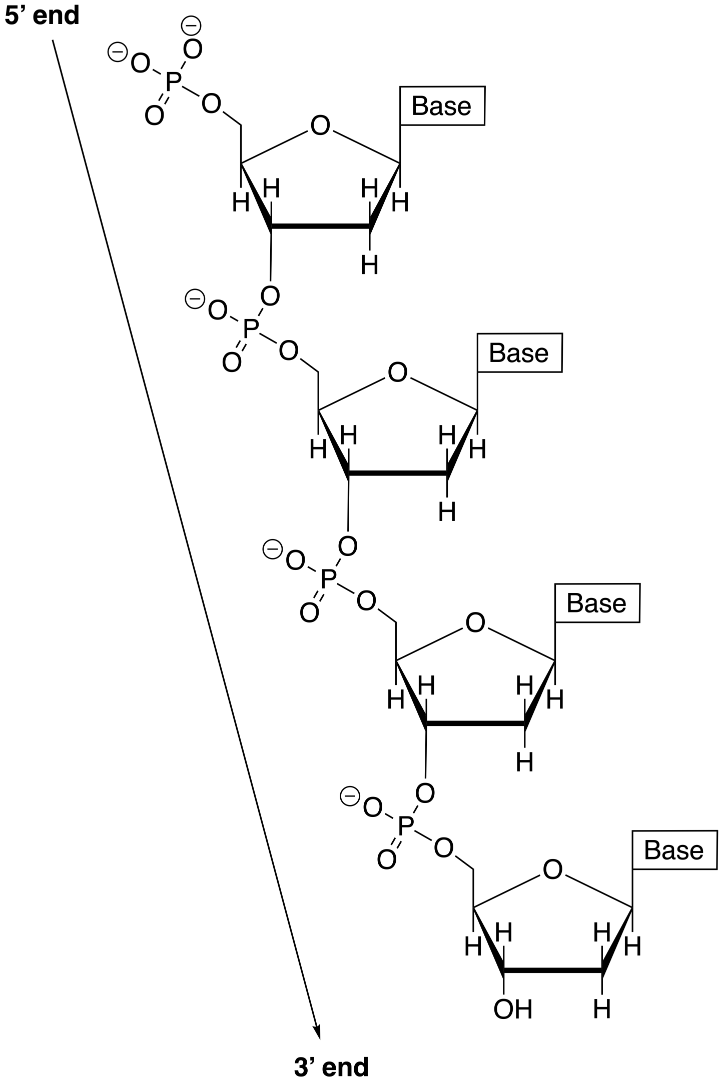

Figure 7: An example of a DNA strand, in this case, with four nucleotides with unspecified nitrogenous bases. Note the phosphodiester bond between PO_4 group and the 3' carbon of different nucleotides; this linkages are the backbone of a polynucleotide strand.

Chapter 5

Constructor

5.1 Summary

Like for the Constructor for the “Nucleotide” class, a Construtor Method is a method that is called during the creation of an object (instatiation process) by the *main* class.[3] It performs a function to allow the new instant of the object “Strand” to be created with its field defined. It must be called every time a new “Strand” Object needs to be created.

Constructor and Description
Strand () Create an empty Strand without specifying any Nucleotide the Strand contains
Strand (*String* str) Create a Strand with the sequence of Nucleotide specified as a *String*
Strand (*char* c) Create a Strand with a single Nucleotide defined as a single *char*

Strand (*char*[] charArray)

Create a Strand with the sequence of Nucleotide specified as an array of *char*

Strand (*ArrayList* <Nucleotide> nucArrayList)

Create a Strand with the sequence of Nucleotide specified as an *ArrayList*

5.2 Constructor Detail

Strand ()

Create a Strand without specifying any nitrogenous base. This will result in the Strand that will only have one blank (’-’) nucleotide.

Strand (*String* str)

Create a Strand with the sequence of Nucleotides specified as a *String*. The *String* can contain any combination of characters. It should be noted that any *String* with any combination of characters is accepted, therefore, one must not have any other character in the input *String* apart from ’*A*’, ’*C*’, ’*G*’, ’*T*’, or ’*N*’; otherwise, the program may crash at later stages (see Chapter 9 for Limitations of the Current Release).

Parameters:
str - A *String*

Strand (*char* c)

Create a Strand with a single Nucleotide defined as a *char*. Any *char* is accepted, however, to ensure that the program doesn’t crash at later stages, one must make sure the characters are either ’*A*’, ’*C*’, ’*G*’, ’*T*’, or ’*N*’ only (see Chapter 9 for Limitations of the Current Release).

Parameters:
c - A *char*

Strand (*char*[] charArray)

Create a Strand with the sequence of Nucleotides specified as an $Array$ of $char$. Any $char$ can be in the input $Array$ of $char$, however, to ensure that the program doesn’t crash at later stages, one must make sure the $char$ in the $Array$ is either ’A’, ’C’, ’G’, ’T’, or ’N’ only (see Chapter 9 for Limitations of the Current Release).

Parameters:
charArray - An $Array$ of $char$

Strand (*ArrayList* <Nucleotide> nucArrayList)

Create a Strand with the sequence of Nucleotides specified as an $ArrayList$ of Nucleotide. The input Nucleotide must be the Object of type Nucleotide properly constructed using a Constructor Method for a Nucleotide Object.

Parameters:
nucArrayList - An $ArrayList$ of Nucleotide

Chapter 6

Method

6.1 Summary

Modifier and Type	Method and Description
static Strand	**random** (*int* **length**) Give a Strand with a certain number of Nucleotide that contains a completely random sequence of Nucleotides
String	**toString**() Returns the sequence of Nucleotide in the Strand to a String
void	**setComplementary**() Turns every Nucleotide in the Strand to its complementary Nucleotide
Strand	**giveComplementary**() Returns a Strand complementary to the current Strand
Nucleotide	**showNucleotide** (*int* **index**) Returns the Nucleotide Object at a particular index in this Strand
int	**size**() Returns the number of Nucleotide in the current Strand

void	**add** (Nucleotide **n**) Add a new Nucleotide at the end of the Strand
void	**add** (*int* **index**, Nucleotide **n**) Add a new Nucleotide at a particular index and pushes every other Nucleotide forward down the Strand
boolean	**equals** (Strand **s1**) Returns *true* if the current Strand and the input Strand has the same sequence of Nucleotide; otherwise returns *false*
Nucleotide	**get** (*int* **index**) Returns the Nucleotide Object at a particular index in this Strand
int	**findSingleBase** (Nucleotide **n**) Returns the index where the first instance of a Nucleotide is found
int	**findSingleBase** (Nucleotide **n**, *int* **startSearchAt**) Returns the index where the first instance of a Nucleotide is found after a certain index inputted
Strand	**concat** (Strand **s2**) Returns a new Strand that is the result of the current Strand being concatenated to the inputted Strand
Strand	**subStrand** (*int* **startIndex**, *int* **finishIndex**) Returns a new Strand that is a cut version of the current Strand, which includes all the Nucleotide from the first index to the index before the last index
int	**searchPiece** (Strand **s1**, *int* **startSearchAt**) Returns the first index after an input index where the Nucleotide sequence of the input Strand is found on the current Strand
int	**searchPiece** (Strand s1) Returns the first index where the Nucleotide sequence of the input Strand is found on the current Strand

Strand	**returnSwapOrder**() Returns a new Strand that has the Nucleotide sequence order completely flipped
int	**countOccurance** (Strand **s1**) Returns the number of times the Nucleotide sequence of the input Strand is found on the current Strand
ArrayList<Strand>	**cleave** (Strand **s1**, Strand **s2**) Returns an *ArrayList* of Strand containing many Strand that would be created if the current Strand is cleaved by an endonuclease that cleaves between the **s1** and **s2**

6.2 Method Detail

static Strand **random** (*int* **length**)

Returns a new Strand Object with **length** number of Nucleotide in the Strand where all the Nucleotide in the Strand is completely random (can be any of the following: ’*A*’, ’*C*’, ’*G*’, and ’*T*’).

Parameters:

length - An *int*

String **toString**()

Returns the Nucleotide sequence of the current Strand as a *String*. For Adenine, ’*A*’ will be returned, ’*C*’ for Cytosine, ’*G*’ for Guanine, and lastly ’*T*’ will be returned for Thymine.

void **setComplementary**()

Changes every Nucleotide in the current Strand to the complementary Nucleotide. As such, Adenine is change to Thymine and vice versa, and Cytosine is changed to Guanine and vice versa. See Chapter 1, Section 1.2 for information about complementary base.

Strand **giveComplementary**()

Returns a new Strand Object that has all the Nucleotide base complementary to the Nucleotide of the current Strand. For example, if the current Strand is "*AGT*", the returned Strand will become "*TCA*". See Chapter 1, Section 1.2 for information about complementary base.

Nucleotide **showNucleotide** (*int* **index**)

Returns the Nucleotide Object at a particular input **index** on the current Strand. This performs the same function as method **get**(*int* **index**).

Parameters:

index - An *int*

int **size**()

Returns an *int* that represents the size of the Strand – that is the number of Nucleotide the current Strand has.

void **add** (Nucleotide **n**)

Appends the input Nucleotide **n** to the end of the Strand.

void **add** (*int* **index**, Nucleotide **n**)

Appends the input Nucleotide **n** to the **index** of the Strand and shifts every Nucleotide after that index by one position down the Strand.

Parameters:

index - An *int*

n - A Nucleotide

boolean **equals** (Strand **s1**)

Returns *true* if the Strand **s1** has exactly the same Nucleotide sequence as that of the current Strand. Otherwise, returns *false*.

Parameters:

s1 - A Strand

Nucleotide **get** (*int* **index**)
Returns the Nucleotide Object at a particular input **index** on the current Strand. This performs the same function as the method **showNucleotide** (*int* **index**). **Parameters:** **index** - An *int*

int **findSingleBase** (Nucleotide **n**)
Returns an *int* that represents the index of the first instance where Nucleotide **n** appears in the current Strand. **Parameters:** **n** - A Nucleotide

int **findSingleBase** (Nucleotide **n**, *int* **startSearchAt**)
Returns an *int* that represents the index of the first instance where Nucleotide **n** appears in the current Strand, if the search for the Nucleotide **n** starts from index **startSearchAt**. **Parameters:** **n** - A Nucleotide **startSearchAt** - An *int*

Strand **concat** (Strand **s2**)
Returns a Strand Object that is the result of concatenation of the current Strand and the Strand **s2**. For example, if the current Strand is “AGT” and the Strand **s2** is “CGT”, the method would return a Strand with the Nucleotide suquence “AGTCGT”. **Parameters:** **s2** - A Strand

Strand **subStrand** (*int* **startIndex**, *int* **finishIndex**)

Returns a shorter Strand Object that is the result of cutting the current Strand, so that the resultant Strand includes only Nucleotide Objects from index **startIndex** to the end index of (**finishIndex**-1). For example, if the current Strand is "AGTCGT", the **startIndex** is 1 and the **finishIndex** is 4, the method would return a Strand with the Nucleotide suquence "GTC".

Parameters:

startIndex - An *int*

finishIndex - An *int*

int **searchPiece** (Strand **s1**, *int* **startSearchAt**)

Returns an *int* that represents the first instant where the exact Nucleotide sequence of the Strand **s1** is found on the current Strand, after the index **startSearchAt** – that is, no matter how many instances there are of the Nucleotide sequence of the Strand **s1** can be found before the index **startSearchAt**, the method will ignore all instance of those.

Parameters:

s1 - A Strand

startSearchAt - An *int*

int **searchPiece** (Strand s1)

Returns an *int* that represents the first instant where the exact Nucleotide sequence of the Strand **s1** is found on the current Strand, if the search was started at the beginning of the Strand (at index 0).

Parameters:

s1 - A Strand

Strand **returnSwapOrder**()

Returns a Strand Object that has the same Nucleotide sequence as the current Strand with the only one difference being that the order of the Nucleotide sequence is completely flipped. For example, if the current Strand is "AGTCA", this method would return a Strand with the Nucleotide suquence "ACTGA".

int **countOccurance** (Strand **s1**)
Returns an *int* that represents the number of times the exact Nucleotide sequence of the Strand **s1** is found on the current Strand, if the search is started from index 0. **Parameters:** **s1** - A Strand

ArrayList<Strand> **cleave** (Strand **s1**, Strand **s2**)
Returns an *ArrayList* of Strand that represents the pieces/strands of DNA that would be created if the current Strand is added to endonuclease that cleaves between the point between two signal sequence, Strand **s1** and Strand **s2**. How it would function can be best represented with an example. For example, if the current Strand is “AAGCAGTTTGACGGAGTTCTA”, and the endonuclease X cleaves between signal sequence “AG” and “TT” (therefore “AG” is Strand **s1** and “TT” is Strand **s2**), then this method would return an *ArrayList* with the Strand Objects: “AAGCAG”, “TTTGATTGACGGAG”, and “TTCTA”. It should also be noted that this method does not work on a Strand with the order of the sequence swapped, and as such in the previous example, the sequence “TTGA” in the second Strand in the output *ArrayList* is never cleaved. **Parameters:** **s1** - A Strand **s2** - A Strand

Part III

How to Use this Package

Chapter 7

Downloading the Class

To use the classes for DNA sequence processing purposes, the class needs to be downloaded. To download the classes included in this package, visit the website https://Soup.Page. Once there, scroll down the page until the Button for "Computer Codes" appear; click on the button "Computer Code". Once there, select "Biology Code", and at that point, the latest stable release of the package will show up. This documentation is only pertaining to v 1.0.0 – the latest stable release that shows up, depending on the time that the site is accessed, may not be v.1.0.0, as newer releases may be uploaded there by that time. If that is the case, v.1.0.0 can be located by clicking on "Archived Version(s)".

Once v.1.0.0 of the package is located, to download the classes, click on "Nucleotide Class" and "Strand Class" (the order in which the two are clicked does not matter). Depending on your web browser and security settings of your computer, you will be asked if you are sure to download the files, and that the files may be harmful to the computer. Please click "Keep", "Download anyway", or any other variations of this on your computer when prompted.

Quick Sidenote from Developer: *I promise you, there is nothing harmful about these two JAVA^{TM}class files that I wrote myself. I can assure you I did not secretly implant malwares, or have virus in there to infect your computer. At the time of writing this, I am an Undergraduate student – I have a ton of other stuffs to do than writing malware code to infect people's computers. The reason that your computer is being spooked about the class files is because I am not a "verified developer" (whatever the heck that means anyway), and that the JAVA^{TM}class files are the type of files that some very bad people can use to infect your computer. But trust me. Even though, I am not* *verified, I am not that type of people, so just trust me, and have fun using my codes. :)*

Once the class files are downloaded, to use the class files, the class files need to be dragged to the folder when a $main$ class exists, or where a $main$ class will be written or placed later (if unfamiliar about $main$ class, please visit Part I, Chapter 1, or Source [1] for more information).

Chapter 8

Using the Class

To use the class, a main method has to be written with proper header “public static void main(String[] args)”. (if unfamiliar about *main* class, please visit Part I, Chapter 1, or Source [1] for more information). Once the main method is appropriately set up, a new Nucleotide object and/or Strand object can be instatiated (set-up/defined) using one of the constructor methods defined in Part I, Chapter 2 and Part II, Chapter 5. The constructor methods are overridden, so that many types of parameters can be passed into the constructor to define the object.

Once this is done, there are many methods that can be performed on the instantiated Nucleotide/Strand object. For full list of methods, please refer to Part I, Chapter 3 and Part II, Chapter 6. Methods, allow for some specific functions to be performed on the object. Some methods require parameters, which must be inputted into the paretheses, and must be of the correct type as is defined in Methods in Part I, Chapter 3 and Part II, Chapter 6. The methods can either be a *void* method, which means that they don’t output anything and is specifically called by itself to perform a function, or a non-*void* method, which outputs a data, therefore the method call has to be set to equal to something to save the output.

When running the program, the *main* class must be the class that gets run, and this class will use the JAVA^{TM}classes downloaded from Soup.Page to perform different functions.

If the user feel unfamiliar with JAVA^{TM}syntax, needs review of the syntax, or unfamiliar with JAVA^{TM}in general, it is recommended that the user takes a class in introductory JAVA^{TM}prorgamming. These courses are available online through services like EdX^{TM}, Coursera^{TM}, and Udemy^{TM}, just to name a few. If however, the user wants to self-study, the book "*Building Java Programs, 5th Edition*" (ISBN: 978-0135471944) is recommended as a resource along with a self-practice platform Practice-It^{TM}(practiceit.cs.washington.edu), which is available for free from the Computer Science Department at the University of Washington.

Part IV

Limitations of the Current Release and Future Releases

Chapter 9

Limitations of the Current Release

1) The current package can only compute DNA: In the real world, nucleic acid comes in two forms: DNA and RNA (Ribonucleic Acid – a chemical cousin of DNA). Since all organisms have RNA, and many non-organismic pathogen like virus and viroid have RNA genome, it is undenaiable that a program that can compute RNA would be beneficial. Since RNA is mostly single-stranded (unlike DNA that is double-stranded), and RNA does not have Thymine (T) but has Uracil (U) instead[2], DNA is rather different from RNA. Because of these difference, to make the package supports RNA computation, the codes needed to be adjusted.

2) Dideoxynucleic acid is not yet supported: In Sanger's Sequencing, the use of Dideoxynucleic acid (dDNA) is essential as it allows the researcher to create DNA Strands of different lengths as dDNA acts as a chain terminator. From there, a mixture of the different DNA strands of different lengths can be run on a gel, and blotted. Then, the DNA sequence can be read.[2] Because of this, it is of great importance that the package is able to account for this special type of nucleic acid, and its role as a chain terminator in future releases.

3) Only Linear DNA can be computed: Considering that majority of organisms on Earth, Bacteria, have a circular DNA, it is of great importance that the package can compute circular DNA. However, as of current, the package can only compute linear DNA that is present in eukaryote, and the only way to carry out computation of circular DNA is to cleave the circular DNA, so it becomes a linear DNA. Recognizing this limitation, future releases will make this package able to compute circular DNA directly to make it more useful in bacterial research.

4) One of the methods for the Strand Class is a duplicate: In the Strand Class, there is a duplicate method in the Strand class, namely the **showNucleotide** method, which is a duplicate of the **get** method. Future releases will eliminate this error.

5) Other degenerate bases are not yet supported: Since when designing different primers, a non-specific, generic bases called degenerate bases need to be defined, it is rather important that the package is able to account for and carry out computation involving those degenerate bases. These degenerate bases include: weak (W), strong (S), purine (R), pyrimidine (Y), amongst others.[4] Since for example, Y can be either Adenine or Guanine, specific codes need to be adjusted to account for this extra functionality. As of current, the only supported degenerate base is N (any Nucleotide), as such, there are rooms for improvement for future releases.

6) The code for blank base is ('Z') not (-): This is rather the culpation on the part of the developer that didn't research carefully that a gap in the DNA strand or a blank base is 'Z' (for "Zero") and not a dash (-). This will need to be fixed in future releases to comply with formal IUPAC rule.[4]

7) It is rather better not to define Nucleotide directly: In this package, a nitrogenous base of DNA is defined directly as a "Nucleotide". While that is correct biologically, this is not the most effective way in computer programming. To make the coding process more efficient, most of the methods in the current Nucleotide class will be moved to a newly made "Nucleobase" class and have the Nucleotide class extends from it. By doing this, the developer will save time when it comes time to write codes for the RNA-base class in the future, as the developer can simply have the RNA-base class extend the Nucleobase class, instead of having to write the code twice.

8) The code doesn't limit users from inputting invalid Nucleotide: This error can be seen in **set**($char$ **base**) method for the Nucleotide class. In this method, any $char$ that is imputted as parameters (including E,F,O,Q, etc. that doesn't exist as a valid DNA/RNAbase or degenerate base) will be accepted. If a $char$ that doesn't correspond to a base is entered, the method will accept these $char$ but crashes later when doing further computation. Therefore, in future releases, this error will be fixed, so that the **set**($char$ **base**) method will not accept any $char$ that doesn't correspond to any existing base to begin with.

Chapter 10

What Might Possibly Be in Future Releases

1) Functionality to allow the code to compute RNA: As discussed earlier in regards to its importance, this functionality will be in future releases.

2) Dideoxynucleic acid computation: Due to its use in Sanger's sequencing, this functionality will be in future releases. To do so, the Nucleotide class will hold two field variables: what kind of nitrogenous base it is, and whether or not it is a dideoxynucleic acid base. The fact that whether or not a nucleotide is a dideoxynucleic acid base will be used in computation in future releases of the package.

3) Computation of circular DNA: Considering that Bacteria, which constitutes the majority of species on Earth, have circular DNA, the ability to do computation with circular DNA directly is an important feature to add to future releases of the package.

4) showNucleotide method of the Strand class will be removed: Due to it being a duplicate of the **get** method, this **showNucleotide** method will be removed, as discussed earlier.

5) Support for degenerate bases: Due to its importance in primer design, degenerate bases including: weak (W), strong (S), purine (R), pyrimidine (Y), amongst others[4] will be added, along with adjustments of the code to allow for novel computations involving these degenerate bases.

6) The blank base/gap in the DNA Strand will be represented by a 'Z' : The '-' that represents the blank base/gap in the DNA strand will be replaced with a 'Z' in future releases to comply with the IUPAC rule.[4]

7) Nucleotide class will extend Nucleobase class: This is to make the process of programming easier as discussed earlier.

8) The set method in the Nucleotide class will limit what $char$ can be inputted as a base: This is to prevent inputting of certain $char$ as a base that will crash the software later.

9) Other new functionalities will be included: To improve the code, new functionalities will be included. These new functionalities include the ability to string together different pieces of DNA, an important process needed for Shotgun Sequencing.[2]

Bibliography

[1] Monica Pawlan. *Essentials, Part 1, Lesson 2: Building Applications*. Oracle Corporation, Redwood City, California, 1999, https://www.oracle.com/technetwork/java/prog-140388.html

[2] Benjamin A. Pierce *Genetics Essentials |Concepts and Connections, 3rd Edition*. W. H. Freeman and Company, New York City, New York, 2016.

[3] Oracle Corporation. *Constructors*. Oracle Corporation, Redwood City, California, 1999, https://docs.oracle.com/javase/tutorial/reflect/member/ctor.html

[4] University of California Santa Cruz. *IUPAC codes*. University of California Santa Cruz, Santa Cruz, California, 2018, https://genome.ucsc.edu/goldenPath/help/iupac.html