# An Introduction to Object Oriented Programming Through Java: BTE324 Final Projects IEEE LaTex Format

Grant Thorburn, *Member, IEEE,* Student

**Abstract**—This technical paper will serve as an introduction to Object Oriented Programming (OOP), using Oracle's Java programming language as step-by-step examples. The sections of the paper are included in the key terms section. The paper will follow the development of a Human object, a Car object, Larry the Track sprinter, and Seahorses. Each section includes step by step examples, developing better code as the examples continue. In addition, this paper will discuss new advances in OOP, as well as and emerging technologies such as AI and IOT.

**Keywords**—Designing Classes, Inheritance, Abstract Classes, Interfaces, Overriding and Overloading, Visibility Modifiers, Lambda Expressions, Functional Programming vs. OOP, Advances in Programming, Concluding Remarks

## 1 Designing Classes

At the core of OOP is the creation of functional objects. A classic example of an object would be a human. A human object has many properties, such as hair color, height, weight, and age. These properties are defined by instance variables, in which give properties to an object. Below is an example of some of the instance variables a human class object could have:

```
public class Human {
        private Double weight;
        private Double height;
        private String color;
        private int age;
```

As noted above, instance variables and classes can contain modifiers, such as "private", that limit the scope of the entity. Modifiers include Public, Protected, and Private. While one does not need a modifier for a variable to be defined, it is good practice to usually instantiate a modifier on a variable. More will be covered in the Visibility Modifiers section.

Instance variables also have data types, as either Primitive or Non-Primitive types. In Object Oriented Programming, one can create an object that can be defined as a data type. If one wanted to create a car class, to be defined within another class, the following example shows a car class and the instantiating of a car object data type:

```
//Car class
public class Car {
//*details of data type Car*
}

//instantiating of Car class
//in another class
private Car nameEx;
//Car is data type
//nameEx is the object
```

In addition, if one wanted to create an ArrayList of data type Car, then within another class this can also be defined as a ArrayList Car object data set. Any Java Collection can be used, such as HashSet, HashMap, Lists, etc. Below is an example of defining an ArrayList Car object data set:

```
private ArrayList<Car> nameEx;
```

Objects also have behavior. In the example of a human, a human can walk, jog, or run. When one plays an online game, objects within the game interact via behaviors in game from user

• *Grant Thorburn is a Business Technology student at the University of Miami, set to graduate in 2019 and become an Army Officer.*
*E-mail: see gat51@miami.edu*

input. These behaviors are defined through methods, which will enact a certain set of actions when called. A calling method will call it's linked method call. To make a human object, we will instantiate "Larry" the human object, an 18 year old male who likes to sprint. Below is an example of a making a Human object, Larry, sprint with his variable maxSpeed:

Human Class:

```
package humanEx;

public class Human {

//Human's variables
private int age;
private String gender;
private double height;
private double maxSpeed;

//Constructor
public Human(int age,
String gender, double height,
double d) {
        super();
        this.age = age;
        this.gender = gender;
        this.height = height;
        this.maxSpeed = d;}
//age, gender, height, and d
//are the parameters of this
//specified construction of
//object Human. All four
//parameters need to be specified
//to call this particular
//Constructor

//Static method.
//Functions when method called
//for a human object.
public static void makeHumanSprint
(double maxSpeed) {
//series of actions, in game,
//in which would make a
//video game character sprint
//to their maxSpeed in game
}

//Getters: public, to return an
//object's unique properties
public int getAge()
{return age;}
public String getGender()
{return gender;}
public double getHeight()
{return height;}
public double getMaxSpeed()
{return maxSpeed;}
}//end of Human class
```

Main Class:

```
public static void
main(String[] args) {
    //instantiate an object Human
    Human larry =
    new Human(18, "Male", 72, 7.4);
        //all four parameters need to
        //be specified Constructor's
        //data type, else an error
//make Human "larry" sprint
Human.makeHumanSprint(
        larry.getMaxSpeed());

}//end of main
```

Within the Main class, Larry the Sprinter is instantiated, assigning the Human object their unique properties. The instantiation in the main class calls the Constructor within the Human class. Larry, as an object, instantiates his own Human class. Think of Larry having his own Human class, defining it for himself to the limit of the classes constructor parameter inputs. Any human can use this class, so long as they are validated with the correct variables. To access Larry's variables, Public Get methods will allow a user to safely call and send variables, regardless of the variable modifier.

If this were to be a video game, having Larry sprint to his maximum speed, then it would be wise to assign a unique identification variable, such as a gamertag. This would allow a system to quickly find a unique human object. Within hospitals, hospital ID's serve to quickly identify a unique patient. If a hospital has thirty John Smith's, this protects each John Smith as unique entities with unique identification. You wouldn't want to perform surgery on the wrong John Smith. In a video game, you want Larry to sprint to his own inherent maximum speed.

## 2 INHERITANCE

As Larry the sprinter is a Millennial, he thinks that he is above being just an average, instantiated human, and deserves his own class, titled "Larry." Java can uniquely cater to Larry's needs by inheriting a superclass (Ex. Human), to a subclass (Ex. Larry), thus utilizing a Polymorphism approach. The most common use of polymorphism in OOP links a parent class to a child class. Although perhaps Larry is on to something: what makes Larry, Larry? All humans share certain traits, such as breathing. In addition, while Larry is able to sprint, unfortunately not all humans can sprint. Logical errors in relationship assumptions can lead to errors as code develops. If a method requires makeHumanSprint(), and a human instantiation does not have a maxSprint speed, the error will continue throughout linked methods. Before writing code, it is important to logically relate an object's inherent, shared properties. In addition, having specialized subclasses help to simplify a portfolio of objects, rather than having one complex Java class.

The Larry class extends the Human class, written on the class line. By inheriting a superclass of Human, the Larry class will inherit all the variables contained within the Human superclass. This is done by calling super(inherited variables) as the first line of the subclasses constructor. is important to note that the Human class is protected from the Larry class. The Human class is protected from the specific nature of Larry's class. Below is a partial code example to show inheritance, modified by moving the maxSpeed method to the Larry class.

```
public class MainClassHeader {
public static void
main(String[] args) {
        Human averageHuman =
        new Human(18, "Male", 72);
        Larry larry =
        new Larry(18, "Male", 72,
        7.4);
        //can even keep generic
        Human larryGeneric =
        new Larry(18, "Male", 72,
        7.4);
        }//end of main
}//end of the Class MainClassHeader

public class Larry extends Human {
        private double maxSpeed;
        public Larry(int age,
        String gender, int height,
        double maxSpeed) {
                super(age, gender,
                height);
                this.maxSpeed =
                maxSpeed;}

        public double getMaxSpeed() {
        return maxSpeed;
        }
        public static void
        makeHumanSprint(double
        maxSpeed) {
        //actions
        }}
```

In a video game, a character may have unique methods and properties within their own classes, while still inheriting the general specifics of a parent class. In the FIFA game series (by EA), the players may have unique dance move methods within their own class, while still inheriting Soccer Player properties.

If we really desired to simplify code, we could make a Sprinter Class extend from Human, and then have a Larry class extend the Sprinter class. Human-Sprinter-Larry relationship would allow other unique Humans, who can Sprint, to share the general mechanics of sprinting.

```
public class Sprinters extends Human {
private double maxSpeed;
public Sprinters(int age,
String gender, int height,
double maxSpeed) {
        super(age, gender, height);
        this.maxSpeed = maxSpeed;}
        public double getMaxSpeed() {
        return maxSpeed;}
        public static void
        makeHumanSprint
            (double maxSpeed) {
    }}
public class Larry extends Sprinters {
        //something that makes
```

```
        //Larry, Larry.
        public Larry(int age,
        String gender, int height,
        double maxSpeed) {
                super(age, gender,
                height, maxSpeed);}
//add unique factor to constructor,
//with appro. getters and methods
}
```

## 3 Abstract Classes

What if all Sprinters would like to make a celebration dance, carrying the general dynamics of dancing, but each Sprinter defines their own dance? Java can implement an Abstract Class, having the Sprinter class as Abstract, allowing all Sprinters to define their own celebration dance. Note the use of public abstract void celebrateDance(); within the Abstract sprinter class:

```
public abstract class Sprinters
extends Human {
//same variables, constructor and
//getter, as previous example, with
//addition of:
public abstract
    void celebrateDance();}

public class Larry
extends Sprinters {
//something that makes Larry, Larry.
public Larry(int age, String gender,
int height, double maxSpeed) {
        super(age, gender,
        height, maxSpeed);}
public void celebrateDance() {
        //dance specific actions. }
}

public class MainClassHeader {
public static void main(String[] args){
//can keep generic, correct:
Human larryGeneric = new Larry(18,
"Male", 72, 7.4);
//Incorrect! Cannot declare
//for a Abstract class:
Sprinter larryGeneric = new Larry(18,
"Male", 72, 7.4);
}//end of main
}//end of the Class "MainClassHeader"
```

Although objects cannot be made from abstract classes, you can still have concrete methods and constructors be inherited from a parent abstract class. Every abstract class method, such as celebrateDance(), must be defined within any object extending from the Abstract class. In addition, in order for behavior to occur, their must be concrete classes inheriting the functions of Abstract classes. Even if a Sprinter chooses not to celebrate by winning a race, every object extending from the Sprinter class can define celebrateDance() for itself. This is the power of OOP polymorphism: classes can share core structures, and yet define methods within their own implementations.

## 4 Interfaces

Interfaces allow a Java developer to separate the functionality of a class from the methods of implementation. An interface is merely a collection of required abstract methods, all left void for implementing classes to define for itself. An implementing class is required to define every interface method. Having interfaces allows a developer to create a shared template to manipulate from. In addition, a java class can implement multiple interfaces, while only being able to inherit from a single parent class.

An important utilization of an interface is the Comparable Interface, provided by the standard Java library to compare two objects. Below is the Comparator Interface:

```
public interface Comparator<T>{
    public int compare(T obj1,
        (T obj2);}
```

Implementing the Comparator will require a compare method within the implemented class, with T standing for equal data types. Comparator interface implementation is very useful to sort Java Collections, as it is customize to sort by comparing to find the greater object of the two. Continuing the Human-Sprinter-Larry example, if we were to desire the Sprinter class to implement specific functions, such as makeHumanSprint, while not having to be an abstract class, creating an interface of Sprinter

will allow this. The Sprinter Interface is implemented by the SprinterImpl code below.

```
public interface Sprinter {
        void makeHumanSprint(
            double d);
}
public class SprinterImpl
extends Human implements
Sprinter{
        private double maxSpeed;

        public SprinterImpl
        (int age, String gender,
        int height, double maxSpeed)
                super(age, gender,
                height);
                this.maxSpeed =
                maxSpeed;}
        public double getMaxSpeed()
        {return maxSpeed;}

        //correct
        public void
        makeHumanSprint(double
        maxSpeed) {
        //unique definition
        //of implemented
        //interface
        }}
```

## 5 Overriding and Overloading

Overloading and Overriding are two distinct programming options that further polymorphic behavior in OOP. Overloading is having two methods with the same name, but different parameters, while overriding is having the same method and parameter combination be inherited to a different function. Overloading provides flexibility to a method, allowing different parameters to cause different method functions, determined by how a method is called (including the parameter set up). Overriding gives inheritance flexibility to inherit a method, and yet "override" it's functionality to the classes specific needs.

A simple overloading example is shown below:

```
//Overloading makeHumanSprint methods
//1) speed as a parameter
public void makeHumanSprint(speed){}

//2) speed and Heart Rate
public void makeHumanSprint(
    speed, heartRate){}

//Overriding is linked to
//parameters called.
//For example, this calls method 2)
makeHumanSprint(5.31, 61);
//This calls method 1)
makeHumanSprint(5.31);
{
//each method has unique impl,
//while still sharing method name
```

It's important to note that you cannot override an implemented interface, as the variables are set in a specific manner, unless you overload the interface as well. Below is an example of both overloading an interface and a method.

```
//interface overloading
public interface Sprinter {
        void
        makeHumanSprint(double d);
        void
        makeHumanSprint(int d);}
//method overloading
public class SprinterImpl
extends Human implements Sprinter{
        private double maxSpeed;
        private int d;

        public SprinterImpl(int
        age, String gender,
        int height,
        double maxSpeed) {
                super(age, gender,
                height);
                this.maxSpeed =
                maxSpeed;
                //cast
                d= (int) maxSpeed;}
        public double
        getMaxSpeed() {
                return maxSpeed;
        }
        //correct
```

```
        public void
        makeHumanSprint
        (double maxSpeed) {
        //unique function
        }
        //also correct
        @Override
        public void
        makeHumanSprint(int d) {
    //also unique function
        }}

//Overriding
```

For an example of overriding, we will take a break from Larry the Sprinter to use Seahorse species as an example. Although each Seahorse species shares a general expected lifespan, as well as general seahorse traits, some species have longer life expectancy's. To simplify an example, we will be using an int variable important that plays a significant factor in determining the expected lifespan of a Seahorse species/subspecies. I am not a Seahorse scientist, so I do not know what's necessarily an important factor. The expectedLife method must be inherited, utilizing a single int variable, as both methods from a parent and child class hierarchy must have the same variable types. The actual variable, important, can be different within each class, so long as they are still the only int variables being implemented. If

```
//Simplified for Readability
Public Class Seahorse{
    int important;
    public void expectedLife
    (int important){
    //unique Seahorse
    //calculations
    }}//end Seahorse

Public Class Zebra_Seahorse
extends Seahorse{}{
    int important;
    public void expectedLife
    (int important){
    //unique Zebra_Seahorse
    //calculations, extended
    //from Seahorse class.
    }}
public class MainClassHeader {
public static void
main(String[] args) {
Seahorse s1 = new Seahorse
(Parameters);
Zebra_Seahorse z1 = new Seahorse
(Parameters);
//s1 calls Seahorse
//expectedLife method
s1.expectedLife(important);
//z1 calls Zebra_Seahorse Method
z1.expectedLife(important);
}}//end MainClassHeader
```

Overriding furthers polymorphic capabilities in programming. In applying programming to genome sequencing, the unique genome of Seahorse's provide ample opportunity for genome scientists to further CRISPR-like concepts and gene editing. Perhaps one day Larry the Sprinter can add gene editing capabilities that were discovered from Zebra Seahorses.

## 6 Visibility Modifiers

Understanding access modifiers, from the inception of a program, is essential to limiting errodious errors. Modifiers are not required, as their is a default modifier. However, ensuring that each relevant variable has the correct scope sets a program up for smooth running. Modifiers can be added to classes, variables, constructors, and methods. The four types of modifiers: Default, Public, Protected, and Private. Visibility of a modified expression is the key factor of modifiers, as the relevant scope between classes, package subclasses, packages, and portfolios will determine which modifier to use. If one has different Human Class implementations within a Java IDE, then ensuring that class calls do not call the wrong "Human" is very important. Larry the Sprinter may end up swimming to his maxSpeed, which may or may not show as an error within the IDE. In addition, it is important to note that just because a variable is visible, it doesn't necessarily mean one can access it.

Before diving into modifiers, it's important to establish what static does for the expression it is modifying. By adding static to an expression, it will make the expression have a single instance.

Depending on the visibility of an expression, it can allow object references to the specific nature of a static variable. Public and Private will help to understand this essential difference of visibility and usability.

Java will use the default modifier when public, private and protected are not specified. The default modifier, in the example of a class, is shown below:

```
Class Zebra_Seahorse(){}
```

If default, it will only be available within the same package. It will not be visible to other packages within a portfolio. The second modifier is public, which makes the modified expression visible throughout a portfolio. There is no restriction on a Public modifier's visibility. A third modifier is Protected, which one can use to limit an expression's visibility to only the same package and differing package subclass. Therefore, a Protected expression can be visible to inherited subclasses within a different package. The final modifier is Private, only visible to the same class. No other class has access to Private modified expressions, regardless of whether or not they are static. Usually within setting up a class, a classes variables should be Private. This ensures that variables are safe from programming errors, such as accidentally calling a variable and changing it. To access a private modifier safely to return, one simply needs to include a Getter. An example is shown below to clarify.

```
//Simplified for readibility.

//Different modifiers with
//the Larry class,
//to be tested within Main class

public class Larry
extends SprinterImpl {
//1)private non-static:
    private int jumpVelocity;
//2)private static:
        private static
        void celebrateDance() {
        //dance specific actions.
        }
//3)public static
        public static
        void verticalJump(){
        //Jump actions
        }
//4)public non-static:
    public void horizJump(){
    //jump actions
    }
//5)protected static,
//returning int
    protected static
    int paceAverage(){
    //pace average actions
    return int_variable;
    }
//6) getter for 1)
    public int getJumpVelocity(){
    return jumpVelocity;}
}//end Larry

//Main Class will show
//visibility, usability,
//and ability to return

public static void main
(String[] args) {
    SprinterImpl a =
    new Larry(xyz);

    //1)cannot call within main

    //2)will not work
    //private not visible:
        a.celebrateDance();

    //3)will work,
    //as static and visible:
        a.verticalJump(xyz);

    //4)visible, not usable.

    //5)visible and usable,
    returning int
    int pace = a.paceAverage();

    //6)visible and usable,
    returning jumpVelocity
    //of an instantiated object
    int velo = a.getJumpVelocity();
```

```
}//end of main
```

## 7 Lambda Expressions

Although C++ and C has had Lambda Expressions for quite some time, Lambda Expressions are new to Java with the Java 8 release. Lambda Expressions utilize functional interfaces, having a single method within the interface. The expression will allow method arguments to be functional, without necessarily having to be within a class. Lambda expressions can also be objects, that can have no parameters, a single parameter, or multiple parameters.

The Lamba operator itself is two parts: left of the dash-arrow, and right of the dash arrow. The left part will transform into the right side, as shown below:

```
//n will double.
n -> 2n
```

A quick example, using the winner of a sprint race, as an interface:

```
interface SprintWinner {
        String winnerInterface
        (String winner);
}

public static void
main(String args[]) {
SprintWinner oneHundredMeterDash
= (String winner) -> "Winner of the
100M Dash is " + winner + "!";
SprintWinner fourHundredMeterDash
= (String winner) -> "Winner of the
400M Dash is " + winner + "!";

//Larry wins the 100M dash
System.out.println(
oneHundredMeterDash.winnerInterface
("Larry"));

//Grant wins the 400M dash
System.out.println(
fourHundredMeterDash.winnerInterface
("Grant"));


}//end main
```

One needs to ensure that their IDE is set up for Java 8 to work. The String winner would be inherited from a method in which found who would win the race. A race method could return winner string to insert into the Lambda Expression. The String Winner will transform into a winner sentance when the SprintWinner interface, winnerInterface, defined the 100M and 400M dashes.

## 8 Difference Between OOP and Functional Programming

Functional languages have a results oriented design: the language is more so about expressing results, rather than showing how those results were generated. This is accomplished by giving an expression to specify an end-result object. The intent is to have software that avoids a mutable data shared state, otherwise known as when a state on multiple threads that share a function or a data structure [**Patrice**]. It is difficult to work with multiple threads on an OOP premise, and functional languages try to eliminate side-effects, known as when Input Output modifies a variable without stating it to the user. OOP and Functional Programming also differ in ease of learning, as Functional Programming seems a bit more jargon-based over learning, say, Python or Java. Although a caveat is that if a proficient programmer uses Functional Programming, it is definitely more concise in it's results driven approach. In addition, as code becomes more complex given Artificial Intelligence, it will be important to modularize and organize code effectively. Functional Programming may have the upper-hand in the future as code becomes more complex.

## 9 Advances in Programming

As of this technical paper's publishing date (May 2018), Artificial Intelligence has been quite the buzzword. Although technology develops quickly, technology does not as quickly implement into society as Moor's Law would suggest. AI research actually began in 1959, as IBM research Arthur Samuel created a self-learning checkers program [**Jones**]. This program was created with binary machine language. However, AI really began its application

in the 1990's, as AI moved from LISP languages (based in machine language lambda calculus) to a diverse set of AI-implementable languages, such as R, Python, AIML (Artificial Intelligence Modeling Language), and even Java.

A fascinating AI programming development is its capabilities for understanding extremely complex neural networks within the human brain. Neurons within the brain fire through synapse connections, linking a synapse to a receptor [**Castronuis**]. Messages are sent and received through the gap when a neuron signal is greater than a certain threshold, being activated and sent to another neuron, as noted by Alex Castrounis of InnoArchitech. Senses such as taste, pain, chills, are all instantaneously sent through the neural network to correctly understand and sense. Implementing AI into Neural Networks is an attempt to recreate the optimization of a human body, within a machine learning premise. Handling neural networks is extremely complex, as inputs can be sent non-linearly, having hidden layers between input and output similar to the Synapse gap. Artificial Neural Networks (ANN) optimizes the weights of neural messages in order to optimize its intended output. The intent is for AI to try and find an optimal solution in the leanest sense of resource utilization. However, AI is also being used to bring to light the "black box algorithms" of the actual calculations, as insights may provide clarity to deploy for solutions. ANN AI applications to Neural Networks allow more powerful insights, as a better AI program can more efficiently utilize a machine learning Neural Network.

Artificial Intelligence is already being established within the programming domain, lead by software initiatives such as the Defense Advanced Research Projects Agency PLINY program, reported in Gartner's "How We Will Work in 2027" published research. The PLINY software, named after Pliny the Elder, who wrote tremendous Roman Encyclopedias, is being designed to write code using AI. The PLINY program can already generate basic coding functions, running automatically. As the program, as well as other AI programming software, gain more access to code to learn from, it is within the realm of possibilities that coding may be AI automated/managed by 2027, as Gartner suggests.

IOT (internet of things) poses tremendous benefits, and possibly dire consequences if not executed correctly. For Internet of Things, a mesh internet network is created with connected embedded sensors, processors and communication hardware. I have worked in commercial real estate, and even within my time working as a Commercial Real Estate Broker, IOT solutions to office buildings was seen as a fascinating technology to implement. Imagine the classic dilemma of street-parking in a city: a choice between choosing a parking spot available far way, or continuing to drive into a city in hopes of finding a spot. Given IOT technology, city's can become "smart cities" by optimizing parking solutions, based on a multitude of variables such as traffic (compared to historic data), event data, as well as the users themselves. However, IOT poses as a tremendous cyber security issue, as IOT devices tend to have weak algorithmic defenses to hackers. Some IOT devices even use "password" as the password. Moving forward into the future, strengthening the security protocols of IOT devices is a lucrative pursuit to create a more information-rich environment.

## 10 Concluding Remarks

This paper serves as an introduction to Object Oriented Programming, covering some of the essentials someone new to Java and OOP could understand. The intent of using Larry the Sprinter was to bring a bit of humor (hopefully it has been somewhat funny, otherwise I have failed miserably) to a challenging subject, as a means of holding the reader's attention for the paper. Although a technical paper could be written like a robot, for an introduction to OOP, I believe Larry serves well. [utf8]inputenc dirtytalk

Their is no shortage of opportunity for students who are both qualitatively and quantitatively qualified. Capegemini consulting firm noted that in 2018, a digital skills gap is impacting 54 percent of all businesses [**Walker**]. Understanding that the technology world moves at as rapid as a pace as Moor's law, I appreciate

that as Students of BTE324, we were expected to figure things out on our own. Within Gartner's How We Will Work in 2027 application, a key conclusion is that most employees will need to continue to hone new skills for the rest of their careers. A meaningful quote to summarize this idea comes from Alvin Toffler, a famed Futurist, in stating that "the illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn." Constantly "upskilling", as Garnter suggests, is something that everyone in the workforce will have to adapt to.

One truly needs to write code every day to understand OOP and Java. I learned to utilize Java's polymorphic capabilities to write more flexible code, thinking through the problem with a user's intentions. This semester, I made a great effort to write code every day. Sometimes I would hit a wall in my code, but instead of getting angry and not address the problem for a while, I learned to manage the inherit walls that coding provides, and manage my effort well to be continuous in attempts. I have GitHub and StackOverFlow to thank for helping me through my issues, one forum/repository at a time. Although I have yet to truly stare down the rabbit hole of programming, I feel confident that I can continue to advance my skill set.

## ACKNOWLEDGMENTS

## ACKNOWLEDGMENT

The author would like to thank Professor Sayed for a challenging semester. The author has benefited tremendously for the course, and looks forward to continuing to develop his programming skills in the future.

## REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] Castrounis, Alex. *Artificial Intelligence, Deep Learning, and Neural Networks Explained*.InnoArchiTech Provides Consulting, Education, Writing, and Speaking Services Focused on Leveraging State of the Art Advanced Analytics to Transform Data into Value — Founded by Alex Castrounis, InnoArchiTech, 1 Sept. 2016.

[3] Jones, Time *Languages Evolve Based on the Unique Requirements of AI Applications* IBM Cognitive Advantage Reports, IBM Corporation, 18 Sept. 2017.

[4] Patrice, Tomas and Jon Skeet *Real-World Functional Programming* Manning Publications, Nov. 2009,

[5] Walker, James. *Digital Skills Gap Now Impacting 54 Percent of Businesses* Digital Journal, 11 Jan. 2018.

**Grant Thorburn** A hard working Business Technology student concluding his junior year at the University of Miami. Grant has enjoyed the challenges posed for the BTE324 course, and feels that he has significantly developed his capabilities to code. As he is involved in the U.S. Army Reserve Officers Training Course at the University of Miami, Grant is aiming to be selected for a Cyber Operations Officer position within the U.S. Army. From studying History in Prague, Czech Republic, to being selected for a Diplomatic mission to Nepal this summer, Grant plans to enter the Army with a global mindset, as well as being technologically capable of solving difficult international problems for the U.S. Army. Grant looks forward to increasing his skillset with next year's coursework within Business Technology, enrolled in Discrete Mathematics, BTE360, BTE400, and BTE423.