# Introduction

This technical document covers concepts taught within the third module of CS 2050. These concepts include Abstract Classes, Interfaces, Generic Classes and Methods, ArrayLists, Queues, Stacks, LinkedLists, Collection sorting, and Maps. For further understanding refer back to [technical document 1](https://docs.google.com/document/u/0/d/140fjDsepU24RmIcUOX5Y71hxQ3gSBQD9IWc9jEj4Osk/edit) and [technical document 2](https://docs.google.com/document/u/0/d/1EObXr1cN3q1XBoJ0P08wdb_nCF13XJDg6N0sSVUuCIU/edit) for prior information.

# Abstract Classes, Interfaces, and Generics

### Abstract Classes

Abstract classes are classes that are used for inheritance but are not directly called. Within an abstract class are like a template for your classes that inherit from it. The difference between abstract and the concrete classes also go further beyond just not being called, they also have methods that must be overridden by the subclasses that inherit from them unlike concrete classes.

abstract class GeometricObject

{

private String color = "white";

private boolean filled;

private java.util.Date dateCreated;

/\*\* Construct a default geometric object \*/

public GeometricObject()

{

dateCreated = new java.util.Date();

}

/\*\* Construct a geometric object with color and filled value \*/

public GeometricObject(String color, boolean filled)

{

dateCreated = new java.util.Date();

this.color = color;

this.filled = filled;

}

/\*\* Return color \*/

public String getColor()

{

return color;

}

/\*\* Set a new color \*/

public void setColor(String color)

{

this.color = color;

}

/\*\*

\* Return filled. Since filled is boolean, the get method is named isFilled

\*/

public boolean isFilled()

{

return filled;

}

/\*\* Set a new filled \*/

public void setFilled(boolean filled)

{

this.filled = filled;

}

/\*\* Get dateCreated \*/

public java.util.Date getDateCreated()

{

return dateCreated;

}

*@Override*

public String toString()

{

return "created on " + dateCreated + "\ncolor: " + color + " and filled: " + filled;

}

/\*\* Abstract method getArea \*/

public abstract double getArea();

/\*\* Abstract method getPerimeter \*/

public abstract double getPerimeter();

}

### Interfaces

An interface is a class with all abstract methods that can be applied to a subclass of another class. The example I used was the pets interface listed below.

interface Pet

{

void beFriendly();

void play();

}

When this interface is added to an object like bulldog it gives it these methods that it will end up overriding to have its unique response and separates the bulldog from other canines that can’t be pets like a wolf.

class Bulldog extends Canine implements Pet

{

public Bulldog(String name, double weight, String food, String location) {

super(name, weight, food, location);

}

*@Override*

public void eat()

{

System.***out***.println(getName() + " is eating " + getFood() + ".");

}

*@Override*

public void beFriendly()

{

System.***out***.println(getName() + " is wagging its tail happily!");

}

*@Override*

public void play()

{

System.***out***.println(getName() + " is playing with a ball.");

}

}

### Generics

Generics are classes or methods with non-concrete types that can be used to take in different data types when needed. These classes and methods will change into the data type required upon instantiating that class or method. An example of a Generic class with methods is the ArrayList class.

# ArrayLists, Queues, Stacks

### ArrayLists

As stated up above the ArrayList is a generic class. It is similar to that of Arrays that we have discussed before but unlike the Array class, ArrayLists don’t have a set limit which allows them to be added to almost indefinitely(The restriction is the set data type and how much memory your device can handle). When defining it as a generic type the class would be ArrayList<E> but that E can be replaced by different data types. In the example below we are using the ArrayList to hold bird objects.

File input = new File("bird.txt");

Scanner entry = new Scanner(input);

int animalCount = entry.nextInt();

ArrayList<Bird> birdlist = new ArrayList<>();

for(int birds = 0; birds< animalCount; birds++)

{

species = entry.next();

name = entry.next();

swimSpeed = entry.nextInt();

if(species == "penguin")

{

Penguin penguin = new Penguin(species,name,swimSpeed);

birdlist.add(penguin);

}

if(species == "duck")

{

Duck duck = new Duck(species,name,swimSpeed);

birdlist.add(duck);

}

if(species == "sootytern")

{

SootyTern sootytern = new SootyTern(species,name,swimSpeed);

birdlist.add(sootytern);

}

if(species == "ostrich")

{

Ostrich ostrich = new Ostrich(species,name,swimSpeed);

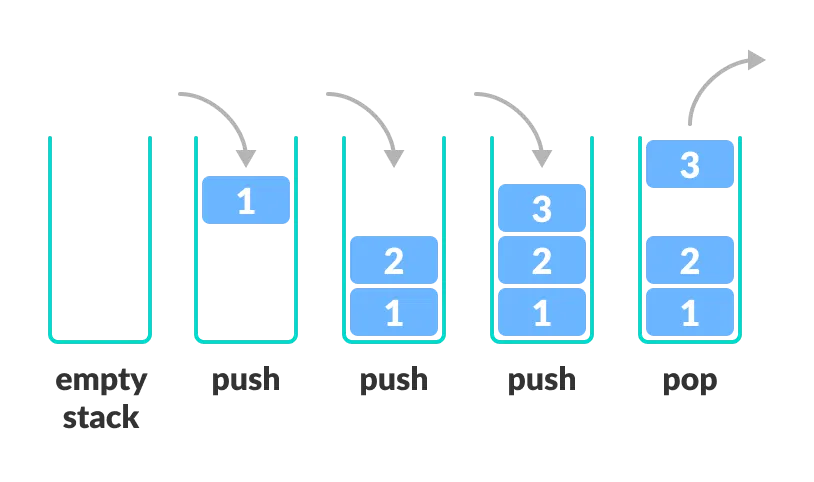
birdlist.add(ostrich);

}

}

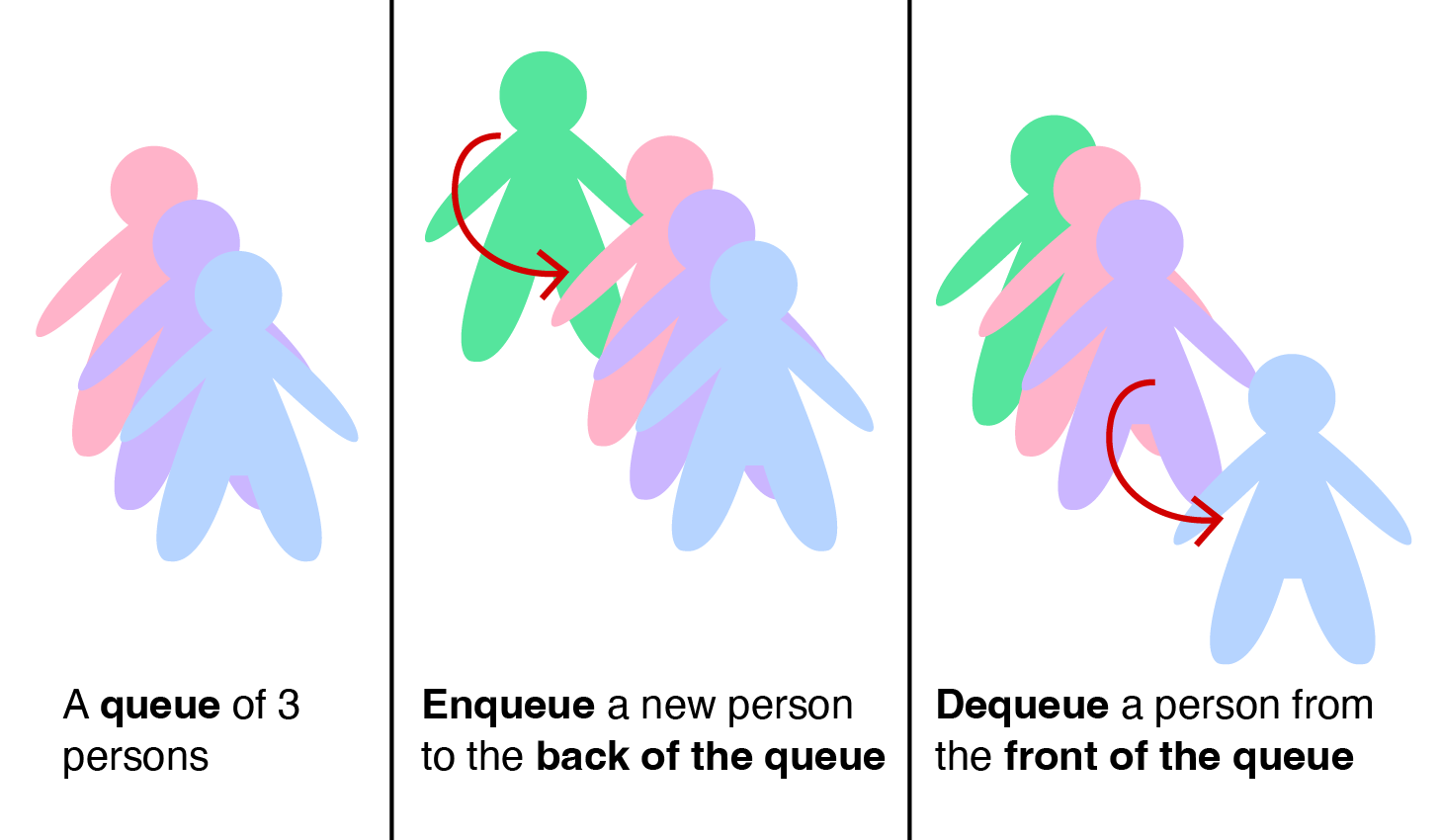
### Stacks

Stacks are a collection of data that can be added to, like an ArrayList, but the information is removed differently than within the ArrayList. A Stack follows a ruling of last in first out. This means as you add items to the stack they can only be removed from the top which is where everything is added in from. The command for adding is known as push and if you want to remove something you pop it out. Below is a visual to represent this process.



### Queues

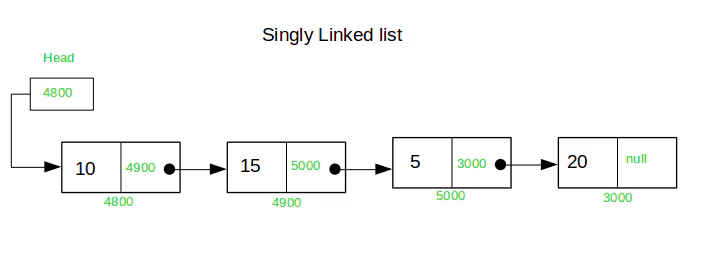
Queues are another collection of data similar to the stack but instead of the last one in being the first one out they do first one in first one out. This allows the information to be processed in the order it was added to the queue. This concept is just like lines at a coffee shop. The first person in line gets served first and then everyone after them has to wait until their turn is up. A visual has been provided below.



# LinkedLists

LinkedLists are collections of data that are stored sequentially and connected together by something called a node. This node allows you to traverse and add items to specific parts of the array list. The node stores the value of an object and has a pointer. There are a few types of LinkedList that we have being Singly, Doubly and Circular.

### Singly LinkedLists



A Singly LinkedList is a type of LinkedList that only has one direction of travel through. You start at the head and move your way to the tail. With this the nodes that are used must have their data value and then a pointer to the one that comes after it. Below is some code for how to insert a node into a Singly LinkedList.

public void insertNode(int number)

{

NodeFix newNode = new NodeFix(number);

//add Temp to hold front of linkedlist

NodeFix Temp = head;

NodeFix current = head;

NodeFix previous = null;

//loop is for counter on list

int loop = 0;

while (current != null && current.data < number)

{

//loops only for inital as that is all the code currently needs.

//Will need change if adding more out of order

if(loop == 0&& head != null)

{

Temp.data= head.data;

}

previous = current;

current = current.next;

//add loop increaser for if statement loops are add more than inital.

loop++;

}

if (previous == null)

{

newNode.next = head;

head = newNode;

}

//changed else into this new if statement to catch reorganizing of LinkedList

if (previous != null)

{

//adds what is past the node to the end of new node and then adds that new node to the end

//of the temp node before adding it to the head.

if(previous.next != null)

{

newNode.next = current;

Temp.next = newNode;

head = Temp;

}

//original else statement

else

{

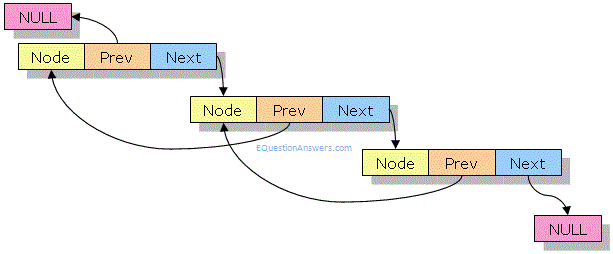
previous.next = newNode;

}

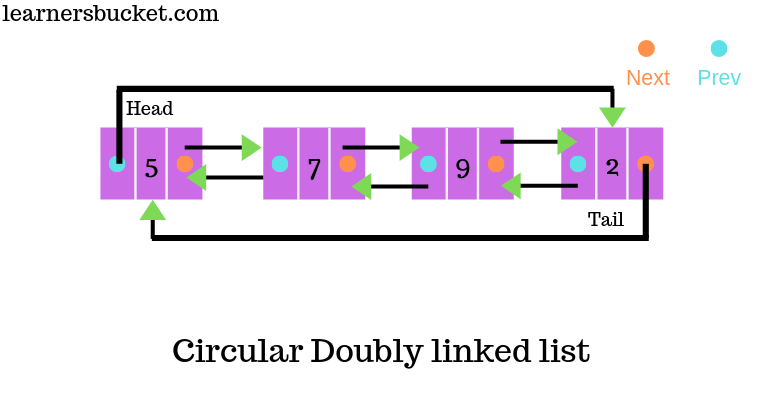
}

}

### Doubly LinkedLists



Now a Doubly LinkedList is similar to the Singly Linked list but it allows travel forward and backwards. This is due to the Node getting an extra parameter for pointing towards the node prior to it. This also allows you to add in new data from both the front and the back of the LinkedList. A more advanced form of a Doubly LinkedList is the Circular Doubly LinkedList. Just like the name suggests the Circular Doubly LinkedList has the last node use its next pointer to point to the beginning of the LinkedList and the first node’s previous pointer point to the end of the LinkedList. Below is a visual of a Circular Doubly LinkedList.



# Collections and Maps

### Collections

Collections are an abstract data type that builds off of Lists(LinkedList and ArrayList). A common use of these collection methods is to use them to sort an ArrayList. These sorting methods use Comparators to compare specifics of objects within those lists to one another and can do it in forward or reverse ordering. Below is a small snippet of code and the results after sorting a list of books by the year they were published.

System.***out***.println("Books sorted by Year:");

books.sort(Comparator.*comparing*(Book::getYear));

for (Book currentBook : books)

{

System.***out***.println(currentBook);

}

System.***out***.println();

Original LinkedList of books:

Unmasking AI by Dr. Joy Buolamwini (2023)

Hello World by Hannah Fry (2018)

The Mathematics of Love by Hannah Fry (2015)

Weapons of Math Destruction by Cathy O’Neil (2016)

Race After Technology by Ruha Benjamin (2019)

Books sorted by Year:

The Mathematics of Love by Hannah Fry (2015)

Weapons of Math Destruction by Cathy O’Neil (2016)

Hello World by Hannah Fry (2018)

Race After Technology by Ruha Benjamin (2019)

Unmasking AI by Dr. Joy Buolamwini (2023)

### Maps

Maps work similarly to collections but require a key in order to access the information. This key is an identifier for the items within the map to quickly find the information the user is requesting. Unlike collections though you can’t have repeating information within a map. Below you can see that the book objects that are in book inventory are being added to bookMapByTitle using a put command and using the identifier of the book being the Title of said book.

*addBook*(bookInventory, userinput);

for (Book book : bookInventory)

{

bookMapByTitle.put(book.getTitle(), book); // key = title, value = book

}

System.***out***.println();

This code then can be used by a searching algorithm to output a book that someone is looking for.

public static void BookSearch(Map<String, Book> bookMapByTitle, Scanner input)

{

String title;

System.***out***.println("Enter the Title of the book you are looking for: ");

input.nextLine();

title = input.nextLine();

System.***out***.println();

if(bookMapByTitle.containsKey(title))

{

System.***out***.println("Found book: " + bookMapByTitle.get(title));

}

else

{

System.***out***.println("Book not found: " + title);

}

}

Enter the Title of the book you are looking for:

Vampirates: Demon of the Ocean

Found book: Vampirates: Demon of the Ocean by Justin Somper (2005)