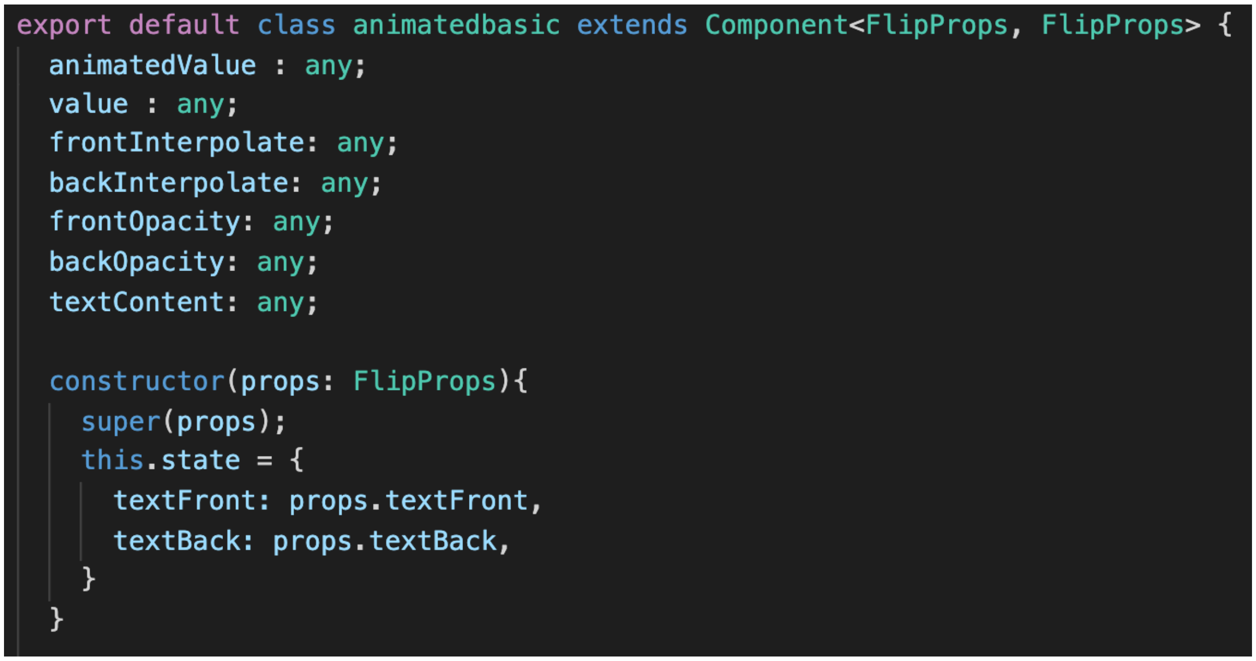
# Criteria C - Development

Techniques Used:

- Encapsulation, Inheritance, Polymorphism
- Functional Programming
- Typescript Types
- XML
- GUI / User Interface

## Encapsulation, Inheritance, and Polymorphism



```
export default class animatedbasic extends Component<FlipProps, FlipProps> {
  animatedValue : any;
  value : any;
  frontInterpolate: any;
  backInterpolate: any;
  frontOpacity: any;
  backOpacity: any;
  textContent: any;

  constructor(props: FlipProps){
    super(props);
    this.state = {
      textFront: props.textFront,
      textBack: props.textBack,
    }
  }
```

Encapsulation, Inheritance, and Polymorphism are some of the most essential parts of an object-oriented programming (OOP) approach. I used some of these techniques in my program so that I could gain some of the benefits of an OOP, even if the entire product doesn't utilize an OOP approach.

Encapsulation is when data is bundled together and can only be operated on or retrieved by using 'getter' and 'setter' methods. The screenshot above shows the declaration of the class "animatedbasic". A number of instance variables are defined for later reference in the class. The "constructor" method is used to define the local variables "textFront" and "textBack" in an instance of the class. These variables cannot be read by any outside function, or changed at all after first defined when an instance of “animatedBasic” is created.

Inheritance allows classes to inherit data structures from other classes. In the class declaration of “animatedbasic” in the same screenshot, inheritance is demonstrated. The “animatedbasic” class “extends” (or inherits) the “Component” class. Inside the “constructor” method, the “super(props)” method calls the constructor method for a “Component” class. This passes the arguments made in the instance of the animatedbasic class (the “props”) to the “Component” constructor.

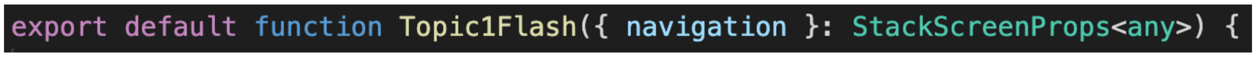
Calling the constructor method of the parent class “Component” ensures that an “animatedbasic” object will have all of the same properties of a “Component” class. In other words, the “animatedbasic” class *inherits* the methods and attributes of the “Component” class. This is also an example of polymorphism, because the constructor is overridden from the parent class. The rest of the “constructor” method in “animatedbasic” goes on to define local variables independent from the “Component” class.

Using inheritance in this context allowed me to create my own components for my product, making development easier. The “Component” class is defined by reactjs. Creating a class that extends, and inherits from the “Component” class, allowed me to easily create a custom component. Without inheritance, I would have had to reinvent the wheel by creating my own version of the “Component” class with the few minor additions that “animatedbasic” adds.

## Functional Programming

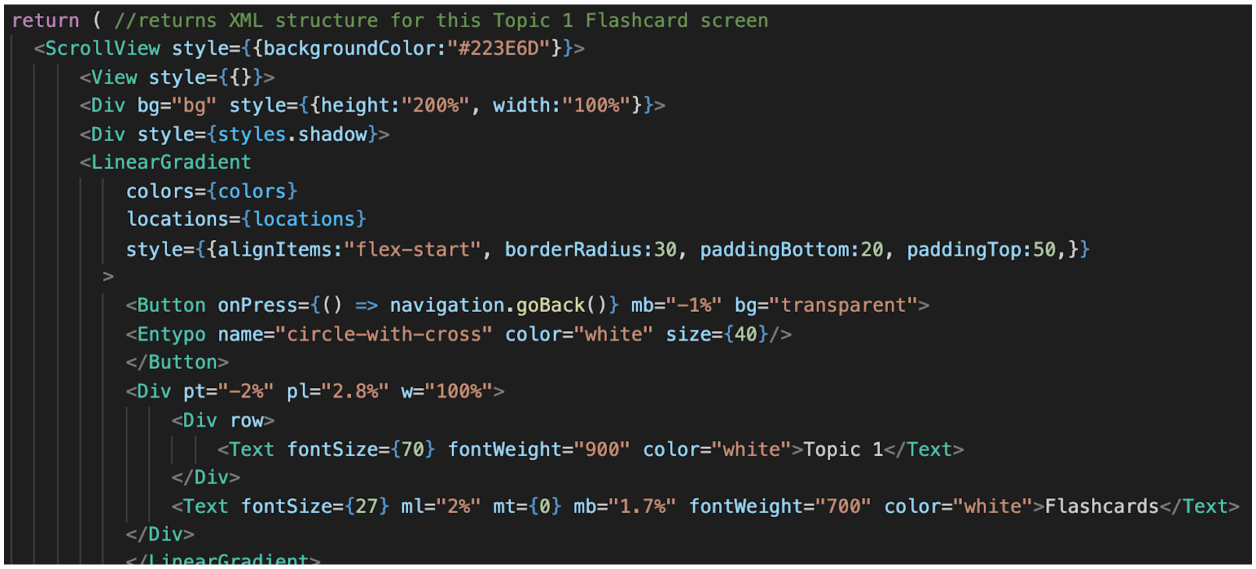
React-native uses a lot of functional programming. Functional programming, in its simplest form, is the development of programs using functions. Because of experience with Java, when first using react-native I expected each screen in the application to be defined by a class. However, most screens in an app when using react-native are actually functions.

For example, the next screenshot shows the declaration of a function “Topic1Flash”. This function represents the entire topic 1 flashcard screen in the app.



```
export default function Topic1Flash({ navigation }: StackScreenProps<any>) {
```

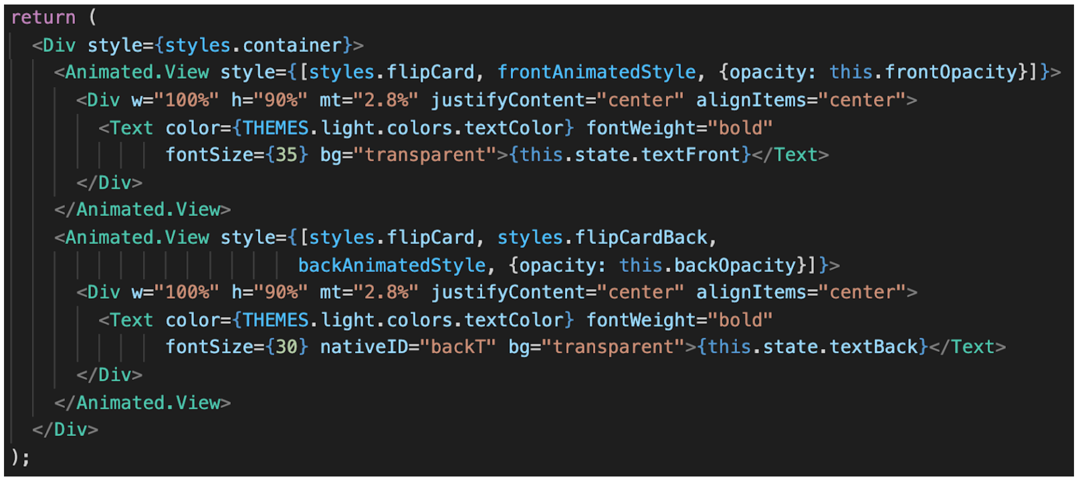
When the function “Topic1Flash” is called, it *returns* the layout and content of the topic 1 flashcard screen in the form of XML data. As shown in the next screenshot below.



```
return ( //returns XML structure for this Topic 1 Flashcard screen
  <ScrollView style={{backgroundColor:"#223E6D"}}>
      <View style={{}}>
      <Div bg="bg" style={{height:"200%", width:"100%"}}>
      <Div style={styles.shadow}>
      <LinearGradient
          colors={colors}
          locations={locations}
          style={{alignItems:"flex-start", borderRadius:30, paddingBottom:20, paddingTop:50,}}
        >
          <Button onPress={() => navigation.goBack()} mb="-1%" bg="transparent">
          <Entypo name="circle-with-cross" color="white" size={40}/>
          </Button>
          <Div pt="-2%" pl="2.8%" w="100%">
              <Div row>
                  <Text fontSize={70} fontWeight="900" color="white">Topic 1</Text>
              </Div>
              <Text fontSize={27} ml="2%" mt={0} mb="1.7%" fontWeight="700" color="white">Flashcards</Text>
          </Div>
          </LinearGradient>
```

This approach of functional programming is similar to object-oriented programming. Each function feels like an object, but the main subtle difference is that this functional programming approach focuses centrally around returning outputs. The main reason for using this approach is because within an app, the content of the screen is sent to the main navigator to be loaded in the application. This action of pushing screen content is simply achieved by placing content in the return statement of a function.

## Using XML



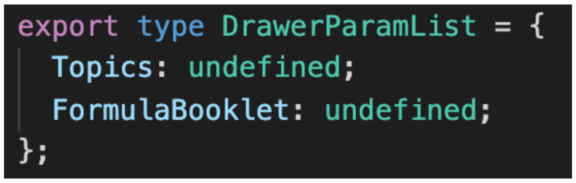
```
return (
  <Div style={styles.container}>
    <Animated.View style={[styles.flipCard, frontAnimatedStyle, {opacity: this.frontOpacity}]}>
      <Div w="100%" h="90%" mt="2.8%" justifyContent="center" alignItems="center">
        <Text color={THEMES.light.colors.textColor} fontWeight="bold"
              fontSize={35} bg="transparent">{this.state.textFront}</Text>
      </Div>
    </Animated.View>
    <Animated.View style={[styles.flipCard, styles.flipCardBack,
                           backAnimatedStyle, {opacity: this.backOpacity}]}>
      <Div w="100%" h="90%" mt="2.8%" justifyContent="center" alignItems="center">
        <Text color={THEMES.light.colors.textColor} fontWeight="bold"
              fontSize={30} nativeID="backT" bg="transparent">{this.state.textBack}</Text>
      </Div>
    </Animated.View>
  </Div>
);
```

XML is a markup metalanguage that allows developers to kind of design their own markup language. Very similar to HTML, XML allows for greater flexibility when designing layouts. XML is a convenient way to format objects. The screenshot above shows an example of a return method returning an XML layout. Each tag like “<Text>”, “<Div>”, or “<Animated.View>” is a component, usually defined by a class with the same name as the tag.

Each of the attributes defined in each component are passed through to the class as properties and arguments. This approach to layout design is easy to quickly understand visually. Because of these benefits, XML is the standard strategy for designing layouts with react-native.

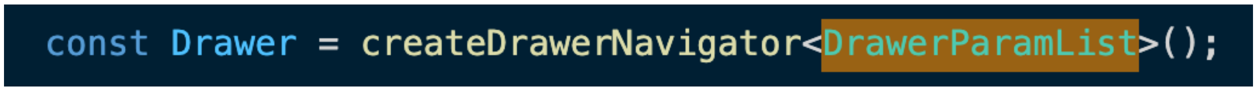
## Typescript Types

When programming applications, data structures can quickly become very complicated. The program needs to keep track of information like page content, page layout, the order in which components are rendered, navigation history, and more. One way to more efficiently deal with complicated data structures is to use *types*. Types are similar to Objects; they have fields, but unlike Objects they do not include any methods. Using types in Typescript allows you to compose custom types using other existing types. In the screenshot below, a custom type "DrawerParamList" is created, built as a composite of two properties "Topics", and "FormulaBooklet", both with type 'undefined'.

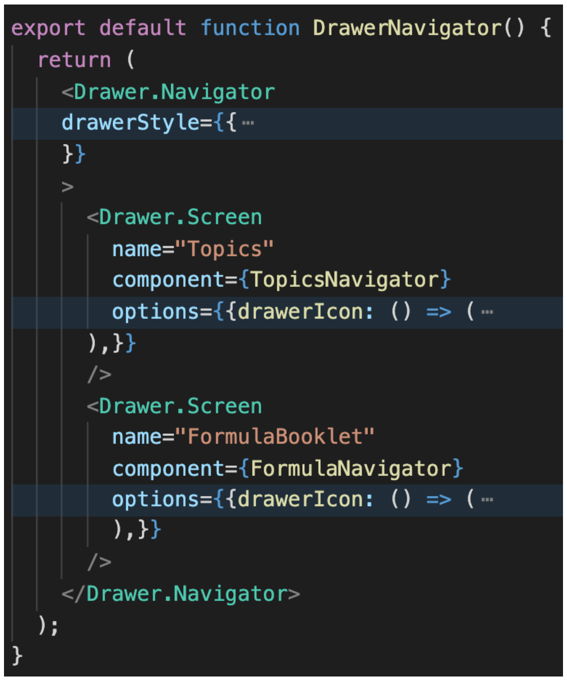


```
export type DrawerParamList = {
  Topics: undefined;
  FormulaBooklet: undefined;
};
```

By creating this custom type, it can be easily applied to a function. The next screenshot below shows the creation of a 'Drawer' using the method "createDrawerNavigator" with a type parameter of the custom type "DrawerParamList". This means that "Drawer" will have two properties: "Topics" and "FormulaBooklet" (each with type 'undefined').



```
const Drawer = createDrawerNavigator<DrawerParamList>();
```



```
export default function DrawerNavigator() {
  return (
    <Drawer.Navigator
    drawerStyle={{ ⋯
    }}
    >
      <Drawer.Screen
        name="Topics"
        component={TopicsNavigator}
        options={{drawerIcon: () => ( ⋯
      ),}}
      />
      <Drawer.Screen
        name="FormulaBooklet"
        component={FormulaNavigator}
        options={{drawerIcon: () => ( ⋯
        ),}}
      />
    </Drawer.Navigator>
  );
}
```

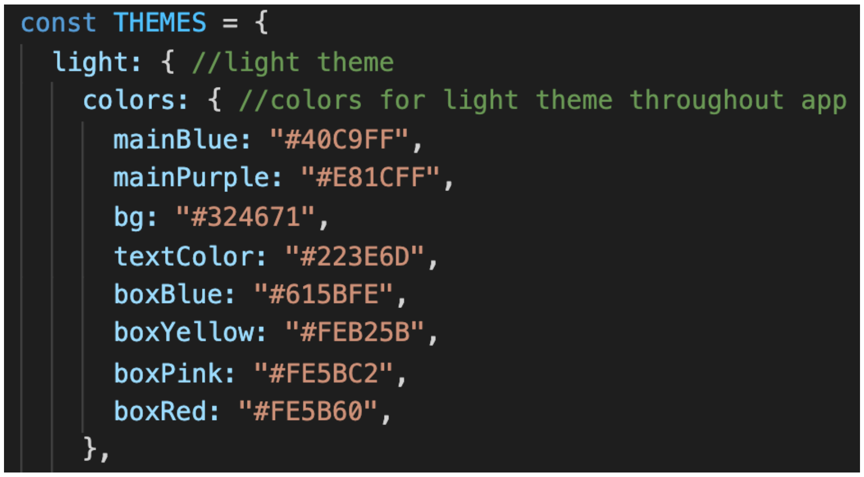
Now, when the navigation drawer is created with the function "DrawerNavigator", the const "Drawer" with the type "DrawerParamList" is used (shown in the left screenshot). Two components "Topics" and "FormulaBooklet" are created. This works because the "Drawer" contains those two properties.

The components "Topics" and "FormulaBooklet" are of type "undefined" so that they can be referred to even before they are loaded into the app (which is, in a way, asynchronous programming).

## GUI / User Interface

Creating an aesthetically pleasing application is important if you want the application to be accessible and approachable to the most number of users. A great way to follow by example when creating clean user interfaces is to look towards Apple's Human Interface Guidelines, designed to help create a standard set of guidelines to follow when designing an application to be used by humans, especially through Apple devices. Apple recommends developers follow these guidelines to make usable, legible, and good-looking applications. One way I followed these guidelines is by always using a font size larger than 17pt, the minimum recommended font size by the Human Interface Guidelines.

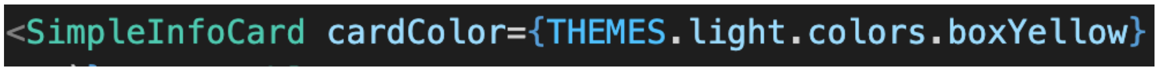
Another aspect of the guidelines is to avoid using colors that make it difficult for users to perceive content in the application. Due to the immense number of components in an application, it is often hard to keep track of every component's color. This creates the risk that a color setting might be forgotten, and something in the app will become illegible as a result.



```
const THEMES = {
  light: { //light theme
    colors: { //colors for light theme throughout app
      mainBlue: "#40C9FF",
      mainPurple: "#E81CFF",
      bg: "#324671",
      textColor: "#223E6D",
      boxBlue: "#615BFE",
      boxYellow: "#FEB25B",
      boxPink: "#FE5BC2",
      boxRed: "#FE5B60",
    },
```

To avoid this, I create a data object "THEMES" which centrally defines most colors throughout the application, shown in the above screenshot. The group "colors" defines hex color codes to be used throughout the code, so everything stays consistent.

The screenshot below shows how the "THEMES" values are referred to in an argument of a component within the application. Changing the value of "boxYellow" in "THEMES" will change the value of "cardColor" in "SimpleInfoCard".



```
<SimpleInfoCard cardColor={THEMES.light.colors.boxYellow}
```

## References

Ahamed, Foysal. "Magnus UI Tutorial: Building Out React Native UI Components - Logrocket Blog". Logrocket Blog, 2021, https://blog.logrocket.com/magnus-ui-tutorial-building-out-react-native-ui-components/.

Chiarelli, Andrea. "The Functional Side Of React." *Medium*. N.p., 2018. Web. 19 Oct. 2021, https://medium.com/@andrea.chiarelli/the-functional-side-of-react-229bdb26d9a6.

"Getting Started". Magnus-Ui.Com, 2021, https://magnus-ui.com/docs/getting-started/.

"Kathawala/Expo-Hamburger-Menu-Template". Github, 2021, https://github.com/kathawala/expo-hamburger-menu-template.

"React Native Template With Hamburger Menu / Drawer Style Navigation". React Native Example For Android And Ios, 2021, https://reactnativeexample.com/react-native-template-with-hamburger-menu-drawer-style-navigation/.

"React Native · Learn Once, Write Anywhere". *Reactnative.dev*, Web. 19 Oct. 2021, https://reactnative.dev/.

Therox, Orta et al. "Object Types." *Typescriptlang.org*. N.p., 2018. Web. 19 Oct. 2021, https://www.typescriptlang.org/docs/handbook/2/objects.html

"Introduction To Expo - Expo Documentation". Expo Documentation, https://docs.expo.io/.

"9 Top App Design Trends For 2021". 99Designs, 2021, https://99designs.com/blog/trends/app-design-trends/.

"Human Interface Guidelines - Design - Apple Developer." *developer.apple.com*. Web. 19 Oct. 2021, https://developer.apple.com/design/human-interface-guidelines/.

"XML Introduction." *W3schools.com*. Web. 19 Oct. 2021, https://www.w3schools.com/xml/xml_whatis.asp.