UNIT - V: ANDROID SERVICES, PUBLISHING ANDROID APPLICATIONS AND IOS

Services, Communication between a service and an activity, Binding activities to services, Threading, preparing for publishing, Deploying APK files iOS tools, iOS project, Debugging iOS apps, Objective-C basics, Hello world app, Building the derby app in iOS.

SERVICES

In Android development, **services** are components that perform background operations, often running independently from an activity. They are typically used for long-running tasks, such as playing music, downloading files, or performing network requests. Services do not provide a user interface. A service can run independently meaning that after your app starts the service, it can run even when your app is no longer in focus.

A Service provides great flexibility by having three different types. They are Foreground, Background, and Bound.

1) **Foreground services**: These services perform tasks that are noticeable to the user, like playing music or handling ongoing notifications.
2) **Background services**: These run without user interaction and without a visible user interface, like syncing data or running in the background for maintenance tasks.

3) **Bound services**: These allow activities (or other components) to bind to the service and interact with it, often for tasks like getting real-time updates or sending commands.

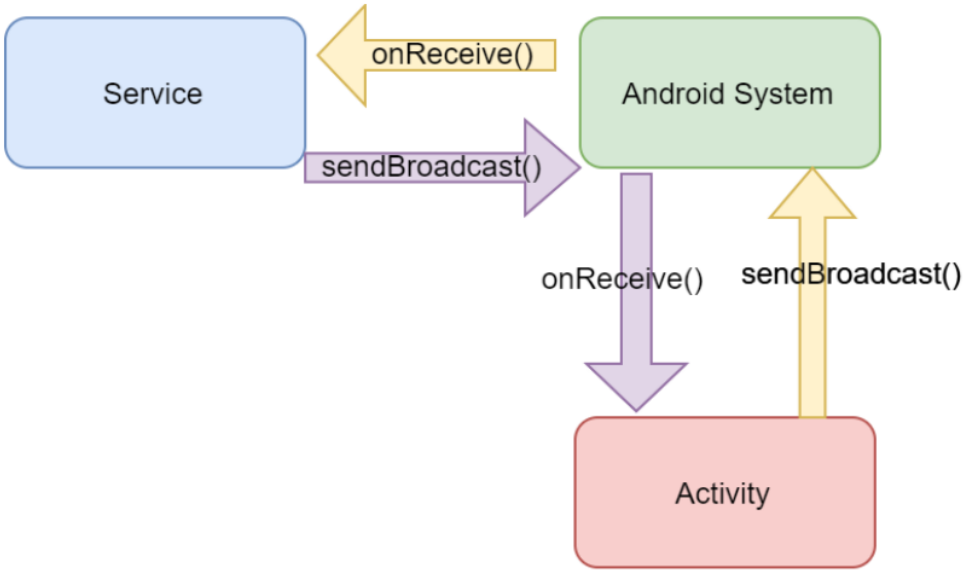
When to Use Service?

We will likely use a service when you want to do something that does not involve UI and needs it to run whether your app is running or not. For example, if you want to play audio a service would be a choice because the audio will still play when the user opens another app.

COMMUNICATION BETWEEN A SERVICE AND AN ACTIVITY

For an activity to communicate with a service, Android provides different mechanisms:

1) **Intents**: You can use explicit intents to start a service. Intents carry data to be processed by the service.
2) **Broadcast Receivers**: Services can send broadcasts that activities or other components listen to and respond to.
3) **Messenger**: A Messenger allows communication using Handler and Message objects. This is useful for passing simple messages back and forth between an activity and a service.
4) **AIDL (Android Interface Definition Language)**: This is used for more complex services, especially when multiple applications need to communicate with a service.



There are two simple ways for you to use an Activity and Service together. One way is to create a bound service and bind a component in your activity to the service acting as a communication channel. Another way is to use the Android Broadcast System, which sends out a broadcast intent system-wide.

BINDING ACTIVITIES TO SERVICES

A **bound service** allows activities to directly interact with the service. This binding is important when the activity requires frequent communication with the service, such as for fetching updates in real time.

Real-time Example: Music Player App

Imagine a music player application where a service is responsible for playing music in the background. The user interacts with the app's UI (the activity), but the music continues playing even when the user navigates away from the activity. In this case, the activity can bind to the service to control the music (play, pause, stop) or retrieve the current playback status.

Step 1: Create the Service-> Define a bound service that plays music

```
class MusicService : Service() {
    private val binder = LocalBinder()
    inner class LocalBinder : Binder() {
        fun getService(): MusicService = this@MusicService
    }
    override fun onBind(intent: Intent?): IBinder {
        return binder
    }
    fun playMusic() {
        // Logic to play music
    }
    fun pauseMusic() {
        // Logic to pause music
    }
    fun stopMusic() {
        // Logic to stop music
    }
}
```

Step 2: Bind the Activity to the Service->The activity can bind and communicate with the service.

```
class MusicPlayerActivity : AppCompatActivity() {
    private var musicService: MusicService? = null
    private var isBound = false
    private val connection = object : ServiceConnection {
        override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
            val binder = service as MusicService.LocalBinder
            musicService = binder.getService()
            isBound = true
        }
        override fun onServiceDisconnected(name: ComponentName?) {
            isBound = false
        }
    }
    override fun onStart() {
        super.onStart()
        Intent(this, MusicService::class.java).also { intent ->
            bindService(intent, connection, Context.BIND_AUTO_CREATE)
        }
    }
```

```
    override fun onStop() {
        super.onStop()
        if (isBound) {
            unbindService(connection)
            isBound = false
        }
    }
    fun onPlayButtonClicked() {
        musicService?.playMusic()
    }
    fun onPauseButtonClicked() {
        musicService?.pauseMusic()
    }
    fun onStopButtonClicked() {
        musicService?.stopMusic()
    }
}
```

- The MusicService is a **bound service** that handles music playback.
- The MusicPlayerActivity binds to the service during its lifecycle (onStart and onStop).
- When the user presses play, pause, or stop buttons, the activity sends commands to the bound service to control the music.

THREADING

In Android, a thread is a unit of execution that can run independently of the main UI thread. Threads can be used to perform long-running tasks without blocking the UI, but they are still part of the same process, not separate background processes.

Threads in Android are typically used to offload tasks from the main thread to avoid freezing the UI. However, they are part of the same application process and share the same memory space. In Java and Kotlin, the Thread class and coroutines can be used to create and manage threads.

Why Threading is Important in Android?

1) **Preventing UI Freezes**: Long-running tasks (e.g., file operations, network requests) can block the UI thread, causing the app to freeze. By moving these tasks to background threads, the UI remains responsive.
2) **Efficient Resource Utilization**: Threading allows multiple tasks to run concurrently, making better use of the device's resources.
3) **Separation of Workloads**: Background tasks like data loading, image processing, or database operations can be separated from UI tasks, improving performance and user experience.

Threading Techniques in Android

1. **Thread and Runnable**
   - The basic approach to threading in Java/Android is using Thread or Runnable.

- A Thread is a unit of execution, and a Runnable is a task that a thread executes.

```
val backgroundThread = Thread(Runnable {
    // Background task, e.g., downloading a file
    downloadFile()
})
backgroundThread.start()
```

2. **HandlerThread**
    - HandlerThread is a subclass of Thread that comes with a Looper. It provides a way to handle messages and run tasks in a background thread.
    - You can use it when you want a dedicated background thread for tasks that will process messages or Runnables.

```
val handlerThread = HandlerThread("BackgroundHandler")
handlerThread.start()
val handler = Handler(handlerThread.looper)

handler.post {
    // Long-running background task
    performLongTask()
}
```

3. **Coroutines (Preferred with Kotlin)**:

- In Kotlin, **coroutines** are the recommended way to perform background tasks, as they are lightweight and provide a simpler, more structured way to handle asynchronous programming.
- Coroutines allow for **suspending functions** that can be paused and resumed without blocking threads.

```
import kotlinx.coroutines.*

fun fetchData() {
    // Start a coroutine in the background
    GlobalScope.launch(Dispatchers.IO) {
        val data = downloadData() // Background task
        withContext(Dispatchers.Main) {
            textView.text = data
        }
    }
}
```

PREPARING APPLICATIONS FOR RELEASE

Preparing your app for release is a multistep process involving the following tasks:

- **Configure your app for release**

  At a minimum, you need to make sure that logging is disabled and removed and that your release variant has debuggable false for Groovy or undebuggable = false for Kotlin script set. You should also set your app's version information.

- **Build and sign a release version of your app**

You can use the Gradle build files with the *release* build type to build and sign a release version of your app. For more information, see Build and run your app.

- **Test the release version of your app**

Before you distribute your app, you should thoroughly test the release version on at least one target handset device and one target tablet device. Firebase Test Lab is useful for testing across a variety of devices and configurations.

- **Update app resources for release**

Make sure that all app resources, such as multimedia files and graphics, are updated and included with your app or staged on the proper production servers.

- **Prepare remote servers and services that your app depends on**

If your app depends on external servers or services, make sure they are secure and production ready. You might need to perform several other tasks as part of the preparation process. For example, you need to create an account on the app marketplace you want to use, if you don't already have one. You also need to create an icon for your app, and you might want to prepare an End User License Agreement (EULA) to protect yourself, your organization, and your intellectual property.

DEPLOYING APK FILES

Release your app to users

You can release your Android apps several ways. Typically, you release apps through an app marketplace such as Google Play. You can also release apps on your own website or by sending an app directly to a user.

Release through an app marketplace

If you want to distribute your apps to the broadest possible audience, release them through an app marketplace.

Google Play is the premier marketplace for Android apps and is particularly useful if you want to distribute your apps to a large global audience. However, you can distribute your apps through any app marketplace, and you can use multiple marketplaces.

Release your apps on Google Play

Google Play is a robust publishing platform that helps you publicize, sell, and distribute your Android apps to users around the world. When you release your apps through Google Play, you have access to a suite of developer tools that let you analyze your sales, identify market trends, and control who your apps are being distributed to.

Google Play also gives you access to several revenue-enhancing features such as in-app billing and app licensing. The rich array of tools and features, coupled with numerous end-user community features, makes Google Play the premier marketplace for selling and buying Android apps.

Releasing your app on Google Play is a simple process that involves three basic steps:

- **Prepare promotional materials**

  To fully leverage the marketing and publicity capabilities of Google Play, you need to create promotional materials for your app such as screenshots, videos, graphics, and promotional text.

- **Configure options and uploading assets**

Google Play lets you target your app to a worldwide pool of users and devices. By configuring various Google Play settings, you can choose the countries you want to reach, the listing languages you want to use, and the price you want to charge in each country.

You can also configure listing details such as the app type, category, and content rating. When you are done configuring options, you can upload your promotional materials and your app as a draft app.

- **Publish the release version of your app**

If you are satisfied that your publishing settings are correctly configured and your uploaded app is ready to be released to the public, click **Publish**. Once it has passed Google Play review, your app will be live and available for download around the world.

iOS Tools

Developing iOS applications requires a suite of specialized tools to streamline the process from design to deployment. Here are some essential tools for iOS mobile application development,

1. **Xcode**
   - Xcode is Apple's official Integrated Development Environment (IDE) for macOS, providing a comprehensive suite of tools for developing, testing, and debugging iOS applications. It includes a code editor, interface builder, and simulators for various Apple devices.
2. **Swift**
   - Swift is a powerful and intuitive programming language developed by Apple for iOS, macOS, watchOS, and tvOS app development. It offers

modern features, safety enhancements, and performance optimizations, making it a preferred choice for developers.

3. **CocoaPods**
   - Cocoa Pods is a dependency manager for Swift and Objective-C projects, simplifying the integration of third-party libraries into Xcode projects. It provides access to a vast repository of open-source libraries, facilitating code reuse and modular development.
4. **Swift Package Manager**
   - Swift Package Manager is a tool for managing the distribution of Swift code. It integrates with the Swift build system to automate the process of downloading, compiling, and linking dependencies, promoting a streamlined workflow.
5. **TestFlight**
   - TestFlight is an online service by Apple that allows developers to invite users to test iOS applications before they are released on the App Store. It supports beta testing, feedback collection, and crash reporting, aiding in the refinement of apps prior to official launch.
6. **Firebase**
   - Firebase is a platform developed by Google offering a suite of tools for app development, including analytics, real-time databases, authentication, and cloud messaging. It assists in building high-quality apps, improving user engagement, and growing the user base.

SETTING UP AN IOS PROJECT

1. **Xcode** is the primary IDE used for iOS development. Start by creating a new project in Xcode using the desired project template (e.g., Single View App, Tabbed App, etc.).

2. **Choose Swift or Objective-C** as the programming language for your project.
3. **Project structure**: Understand the folders and files generated:
   - AppDelegate and SceneDelegate for app lifecycle management.
   - ViewController for controlling the UI logic.
   - Main.storyboard or SwiftUI files for designing the user interface.

iOS Project Structure

1. **AppDelegate.swift:** Manages app lifecycle events (e.g., app launch, background, foreground).
2. **SceneDelegate.swift:** Manages individual scenes for your app (e.g., when using multi-window features).
3. **ViewController.swift**: This is where you'll implement your UI logic.
4. **Main.storyboard**: This file defines your app's visual interface.

AppDelegate.swift

```
import UIKit

@UIApplicationMain

class AppDelegate: UIResponder, UIApplicationDelegate {

   var window: UIWindow?

   func application(_ application: UIApplication, didFinishLaunchingWithOptions
launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

      // Override point for customization after application launch.

      return true
```

```
    }

    func applicationWillTerminate(_ application: UIApplication) {
        // Called when the application is about to terminate. Save data if appropriate.
    }
}
```

SceneDelegate.swift

```
import UIKit

class SceneDelegate: UIResponder, UIWindowSceneDelegate {
    var window: UIWindow?
    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {
        guard let windowScene = (scene as? UIWindowScene) else { return }
        // Programmatically create the main window
        window = UIWindow(windowScene: windowScene)
        window?.rootViewController = ViewController()
        window?.makeKeyAndVisible()
    }
    func sceneDidEnterBackground(_ scene: UIScene) {
        // Save data when the app enters the background.
    }
}
```

ViewController.swift

```
import UIKit

class SceneDelegate: UIResponder, UIWindowSceneDelegate {


    var window: UIWindow?

    func scene(_ scene: UIScene, willConnectTo session: UISceneSession, options
connectionOptions: UIScene.ConnectionOptions) {

        // Use this method to optionally configure and attach the UIWindow `window` to the
provided UIWindowScene `scene`.

        guard let windowScene = (scene as? UIWindowScene) else { return }

        window = UIWindow(windowScene: windowScene)

        window?.rootViewController = ViewController()

        window?.makeKeyAndVisible()

    }

    func sceneDidEnterBackground(_ scene: UIScene) {

        // Save data when the app enters the background.

    }

}
```

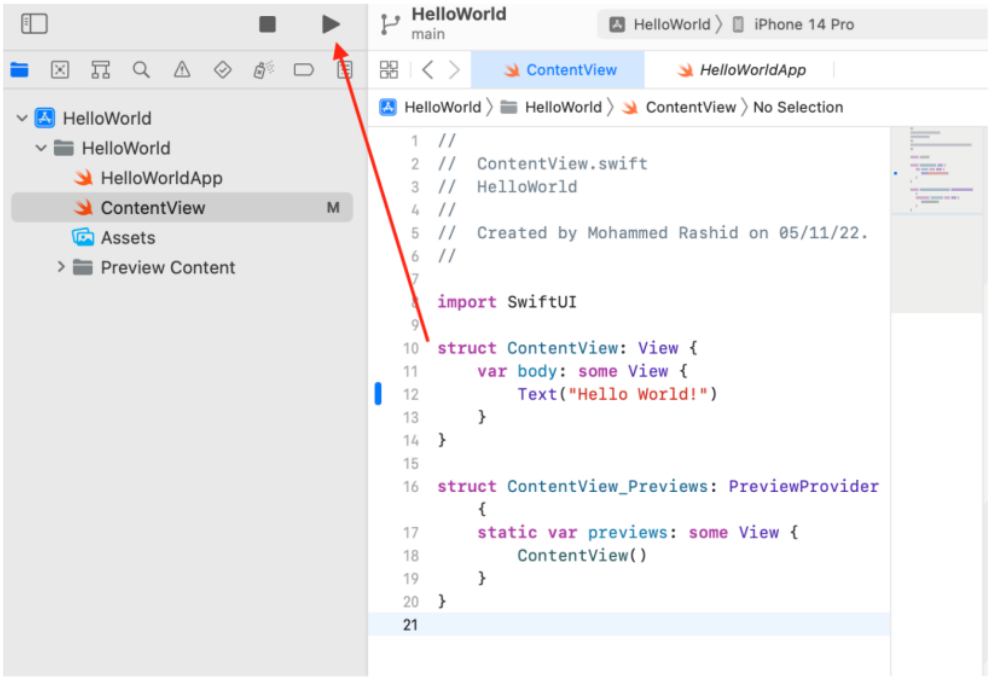
Main.storyboard

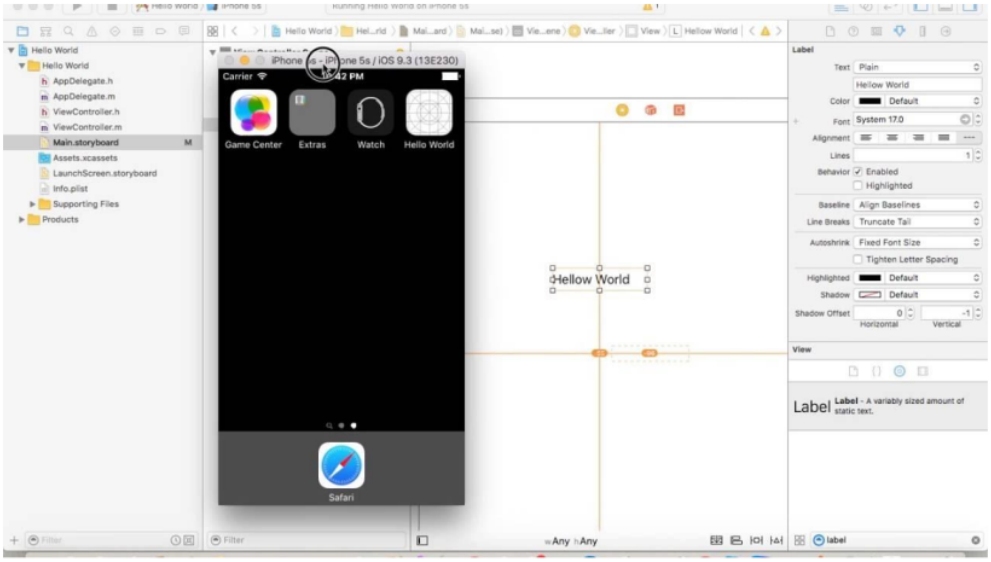
```
import UIKit

class ViewController: UIViewController {
```

```
    @IBOutlet weak var myLabel: UILabel!


    override func viewDidLoad() {
        super.viewDidLoad()
        myLabel.text = "Hello, Storyboard!"
    }
}
```





COMMON DEBUGGING TOOLS AND TECHNIQUES

1. **Xcode Debugger (LLDB)**
   - The LLDB (Low-Level Debugger) is integrated into Xcode and can help track down issues by stepping through code, inspecting variables, and setting breakpoints.
   - Use **breakpoints** to pause execution at certain lines of code. This lets you inspect the state of the app at that moment, including variable values and memory usage.
   - You can view the **console output** to see logs or use LLDB commands for advanced debugging.

2. **Print Statements and Console Logs**
   - Add print() statements to track the flow of the app and inspect variable values at runtime.
   - Use **NSLog()** in Objective-C or **os_log()** for more structured logging.

3. View Hierarchy Debugging

- Xcode provides a powerful tool to debug the **view hierarchy** visually. Use it when you're having layout issues to inspect how views are rendered.
- You can access the view debugger by running your app and clicking the **Debug View Hierarchy** button.

4. Memory and Performance Profiling (Instruments)

- Use **Instruments** (a performance analysis tool bundled with Xcode) to track:
  - **Memory leaks**: Ensure your app is releasing memory properly to avoid crashes.
  - **CPU usage**: Check if there are any performance bottlenecks.
  - **Time Profiler**: Identify which methods consume the most time.
  - **Energy usage**: Useful when debugging performance for apps with background services or intensive computations.

5. Crash Logs and Exception Breakpoints

- Xcode will display **crash logs** if your app terminates unexpectedly. Review the logs to understand what caused the crash.
- Enable **Exception Breakpoints** to catch runtime exceptions as they occur, stopping the app at the point where the error happened.

6. Testing

- Write **unit tests** and **UI tests** using Xcode's built-in testing framework (XCTest) to automate the detection of errors.
- Running tests allows you to identify regressions early in the development process.

7. Simulators

- Run your app on various **simulators** to mimic different iOS devices and screen sizes.
- Simulators allow you to test the app without needing physical devices. However, always test on real hardware before deployment.

BUILDING THE DERBY APP IN IOS

1. Setting Up the Project

- Open **Xcode** and create a new project.
- Select **App** under the iOS tab and click **Next**.
- Name the project "DerbyApp" and choose **Swift** as the language.
- Set the interface to **Storyboard** and leave the other settings as they are.
- Click **Next**, choose a location for your project, and click **Create**.

2. Designing the User Interface (UI)

In **Main.storyboard**, you will create the interface for the Derby game.

a. Add UI Elements

- Open Main.storyboard and select the **ViewController**.
- Drag **Image Views** from the Object Library (on the right) onto the storyboard. Each Image View will represent a horse. Add three Image Views for three horses.
- Set images for the horses (you can use placeholder images for now).
- Add a **UIButton** labeled "Start Race" that will start the race when tapped.
- Add a **UILabel** at the top to display which horse wins the race.

b. Arrange the UI

- Position the three horses near the left edge of the screen.
- Align them vertically with some space between them.
- Place the "Start Race" button below the horses.
- Position the result label at the top.

c. Connect UI Elements to Code

- Open the **Assistant Editor** (click the two overlapping circles at the top right) so that you can see both the ViewController.swift file and the storyboard side by side.
- **Ctrl-drag** from each of the three Image Views to the ViewController.swift file to create outlets. Name them horse1, horse2, and horse3.
- **Ctrl-drag** from the "Start Race" button to create an action method. Name it startRaceTapped.
- **Ctrl-drag** from the label to create an outlet called resultLabel.

3. Writing the Game Logic in Swift

Now, let's implement the logic for the Derby race. The horses will move across the screen when the race starts, and the game will declare a winner when the first horse reaches the finish line.

Open ViewController.swift and update it as follows:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var horse1: UIImageView!

    @IBOutlet weak var horse2: UIImageView!

    @IBOutlet weak var horse3: UIImageView!
```

```
// Outlet for the result label
@IBOutlet weak var resultLabel: UILabel!

// A timer to animate the horses
var raceTimer: Timer?

// Track if the race is running
var raceInProgress = false

override func viewDidLoad() {
    super.viewDidLoad()
    // Initialize the result label to be empty at start
    resultLabel.text = ""
}

// Start race button action
@IBAction func startRaceTapped(_ sender: UIButton) {
    if raceInProgress {
        return // Ignore if a race is already running
    }
```

```
        raceInProgress = true
        resultLabel.text = "" // Clear the result label before the race

        // Reset horse positions
        resetHorsePositions()

        // Start the race timer to move horses
        raceTimer = Timer.scheduledTimer(timeInterval: 0.05, target: self, selector:
#selector(moveHorses), userInfo: nil, repeats: true)
    }

    // Reset the horses to the starting line
    func resetHorsePositions() {
        horse1.frame.origin.x = 20
        horse2.frame.origin.x = 20
        horse3.frame.origin.x = 20
    }

    // Move horses forward
    @objc func moveHorses() {
```

```
    let finishLine = view.frame.width - 100

    // Move each horse by a random amount

    horse1.frame.origin.x += CGFloat.random(in: 2...10)

    horse2.frame.origin.x += CGFloat.random(in: 2...10)

    horse3.frame.origin.x += CGFloat.random(in: 2...10)


    // Check if any horse has crossed the finish line

    if horse1.frame.origin.x >= finishLine {

        declareWinner(horse: "Horse 1")

    } else if horse2.frame.origin.x >= finishLine {

        declareWinner(horse: "Horse 2")

    } else if horse3.frame.origin.x >= finishLine {

        declareWinner(horse: "Horse 3")

    }

}


// Stop the race and declare a winner

func declareWinner(horse: String) {

    raceTimer?.invalidate() // Stop the race timer

    raceInProgress = false
```

```
        // Update the result label with the winner
        resultLabel.text = "\(horse) Wins!"
    }
}
```

Explanation of Code

- **Outlets and Actions**: The outlets connect the UI elements (horses and result label) to the code. The startRaceTapped function is triggered when the user taps the "Start Race" button.
- **Race Timer**: The timer (raceTimer) is used to repeatedly move the horses across the screen. It updates every 0.05 seconds and moves each horse by a random number of pixels.
- **Finish Line**: We define the finish line as view.frame.width - 100 (100 pixels from the right edge of the screen).
- **Random Movement**: Each horse moves forward by a random amount between 2 and 10 pixels, simulating the race.
- **Winner Declaration**: When a horse crosses the finish line, the race stops, and the result label displays the winner.

4. Running the App

- Press Cmd + R or click the **Run** button in Xcode to build and run the app in the iOS Simulator.
- Tap the "Start Race" button to begin the race, and watch as the horses move across the screen.
- The result label will display the winner once one of the horses crosses the finish line.

MOBILE APPLICATION DEVELOPMENT-Full Material.docx

ORIGINALITY REPORT

20%	15%	6%	10%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Swinburne University of Technology Student Paper	1%
2	Submitted to University of Greenwich Student Paper	1%
3	magora-systems.com Internet Source	1%
4	www.insighttycoon.com Internet Source	1%
5	www.coursehero.com Internet Source	1%
6	developers.cyberagent.co.jp Internet Source	1%
7	Submitted to Manipal University Jaipur Online Student Paper	1%
8	Submitted to Glasgow Caledonian University Student Paper	1%
9	fastercapital.com Internet Source	1%

10	www.svec.education Internet Source	1 %
11	Submitted to Melbourne Institute of Technology Student Paper	1 %
12	Submitted to CTI Education Group Student Paper	1 %
13	dev.to Internet Source	1 %
14	Submitted to De Montfort University Student Paper	<1 %
15	Submitted to West Herts College Student Paper	<1 %
16	atlantacrowdsource.com Internet Source	<1 %
17	medium.com Internet Source	<1 %
18	ia802508.us.archive.org Internet Source	<1 %
19	Submitted to Amman Baccalaureate School Student Paper	<1 %
20	Submitted to Manipal University Student Paper	<1 %
21	Submitted to University of West London Student Paper	

		<1%
22	answeregy.com Internet Source	<1%
23	www.theseus.fi Internet Source	<1%
24	Submitted to Liverpool John Moores University Student Paper	<1%
25	Submitted to National Institute of Business Management Sri Lanka Student Paper	<1%
26	Submitted to Queen Mary and Westfield College Student Paper	<1%
27	knowlesti.sg Internet Source	<1%
28	www.fastercapital.com Internet Source	<1%
29	blog.hellomobile.com Internet Source	<1%
30	students.warsidi.com Internet Source	<1%
31	www.sweetstudy.com Internet Source	<1%

32	Margaret Kozak Polk. "Coding Android Apps", CRC Press, 2024 Publication	<1 %
33	Olimpia Giuliana Loddo, Andrea Addis, Giuseppe Lorini. "Intersemiotic translation of contracts into digital environments", Frontiers in Artificial Intelligence, 2022 Publication	<1 %
34	sist.sathyabama.ac.in Internet Source	<1 %
35	www.thenationalnews.com Internet Source	<1 %
36	5app.ru Internet Source	<1 %
37	denver-app-development91245.dsiblogger.com Internet Source	<1 %
38	infogalactic.com Internet Source	<1 %
39	vdoc.pub Internet Source	<1 %
40	www.androidtablets.ca Internet Source	<1 %
41	www.geeksforgeeks.org Internet Source	<1 %

42	www.webdevelopmenthelp.net Internet Source	<1 %
43	Submitted to Staffordshire University Student Paper	<1 %
44	Xamarin Mobile Application Development, 2015. Publication	<1 %
45	open-innovation-projects.org Internet Source	<1 %
46	svec.education Internet Source	<1 %
47	www.droidape.com Internet Source	<1 %
48	www.qfs.de Internet Source	<1 %
49	"ICDSMLA 2019", Springer Science and Business Media LLC, 2020 Publication	<1 %

Exclude quotes	On	Exclude matches	Off
Exclude bibliography	On		