Prácticas y Evaluación Final: Fundamentos de Swift y Programación Orientada a Objetos

Objetivo General: Evaluar la comprensión y aplicación de los conceptos fundamentales del lenguaje de programación Swift y los principios de la Programación Orientada a Objetos (POO) en la resolución de problemas prácticos, utilizando el compilador en línea Replit para la codificación y ejecución de los ejercicios.

Instrucciones Generales:

1. Cada alumno deberá crear una cuenta en Replit si aún no la tiene. (Se puede usar un correo de Gmail para crear la cuenta)
2. Para cada una de las partes, puedes crear varios proyectos/Apps de "Repitl" usando la plantilla de Swift.
3. El código deberá estar bien comentado, explicando la lógica detrás de las implementaciones.
4. El alumno deberá compartir los enlaces de su "Replit" para su revisión.
5. Presta atención a los detalles de cada instrucción para asegurar que se cumplan todos los requisitos.

Parte 1: Fundamentos de Swift

En esta sección, demostrarás tu dominio de los elementos básicos del lenguaje Swift.

Ejercicio 1: Variables, Constantes y Tipos de Datos

- **Objetivo:** Comprender la declaración y uso de variables y constantes, así como la inferencia y asignación explícita de tipos de datos en Swift.
- **Instrucciones:**
    1. Declara una constante llamada `nombreCompleto` de tipo `String` y asígnale tu nombre.
    2. Declara una variable llamada `edad` de tipo `Int` y asígnale tu edad.
    3. Declara una variable `promedioGeneral` e inicialízala con un valor decimal (ej. 8.5). Permite que Swift infiera el tipo.
    4. Intenta reasignar un nuevo valor a la constante `nombreCompleto`. Observa y comenta en el código el error que produce el compilador.
    5. Imprime en consola los valores de `nombreCompleto`, `edad` y `promedioGeneral` utilizando interpolación de cadenas.

Ejercicio 2: Operadores y Expresiones

- **Objetivo:** Aplicar correctamente los operadores aritméticos, de comparación y lógicos en Swift.
- **Instrucciones:**
  1. Declara dos variables numéricas `numeroA` con valor 15 y `numeroB` con valor 7.
  2. Calcula e imprime en consola la suma, resta, multiplicación y división de `numeroA` y `numeroB`. Para la división, asegúrate de que el resultado sea de tipo `Double`.
  3. Declara dos variables booleanas `esMayorDeEdad` (asignando `true` si tu edad es mayor o igual a 18, `false` en caso contrario) y `tieneLicencia` (asigna `true` o `false` según tu caso).
  4. Evalúa e imprime si puedes conducir legalmente (se requiere ser mayor de edad y tener licencia) usando operadores lógicos.

Ejercicio 3: Control de Flujo - Condicionales

- **Objetivo:** Implementar la toma de decisiones en el código utilizando sentencias `if-else` y `switch`.
- **Instrucciones:**
  1. Pide al usuario que ingrese una calificación numérica (0-100) usando `readLine()`. Recuerda convertir la entrada a un `Int`.
  2. Utilizando una estructura `if-else if-else`, imprime en consola el rendimiento del alumno basado en la calificación:
     - 90-100: "Excelente"
     - 80-89: "Bueno"
     - 70-79: "Suficiente"
     - Menor a 70: "Necesita mejorar"
  3. Declara una variable `diaSemana` de tipo `String` y asígnale un día (ej. "Lunes").
  4. Utilizando una sentencia `switch`, imprime un mensaje diferente para al menos 3 días de la semana y un mensaje por defecto para los demás.

Ejercicio 4: Control de Flujo - Bucles

- **Objetivo:** Utilizar bucles `for-in` y `while` para ejecutar bloques de código repetidamente.
- **Instrucciones:**
  1. Utiliza un bucle `for-in` para imprimir los números del 1 al 10.

2. Declara un array de `String` con tus 3 películas favoritas. Utiliza un bucle `for-in` para imprimir cada película precedida por "Me gusta la película: ".
3. Utiliza un bucle `while` para simular una cuenta regresiva desde 5 hasta 1, imprimiendo cada número. Al final, imprime "¡Despegue!".

Ejercicio 5: Colecciones - Arrays y Diccionarios

- **Objetivo:** Demostrar el uso de arrays y diccionarios para almacenar y gestionar colecciones de datos.
- **Instrucciones:**
  1. Crea un array mutable llamado `frutas` que contenga al menos 5 nombres de frutas.
  2. Agrega una nueva fruta al final del array.
  3. Elimina la segunda fruta del array.
  4. Itera sobre el array e imprime cada fruta.
  5. Crea un diccionario mutable llamado `capitales` donde las llaves sean nombres de países (`String`) y los valores sean sus capitales (`String`). Agrega al menos 3 pares país-capital.
  6. Agrega un nuevo país y su capital al diccionario.
  7. Accede e imprime la capital de uno de los países.
  8. Itera sobre el diccionario e imprime cada país junto con su capital.

Ejercicio 6: Optionals

- **Objetivo:** Comprender y manejar de forma segura los valores opcionales en Swift.
- **Instrucciones:**
  1. Declara una variable opcional `posibleNumero` de tipo `String` y asígnale el valor "123".
  2. Utiliza `if let` para desempaquetar de forma segura `posibleNumero`. Si tiene un valor, conviértelo a `Int` e imprímelo. Si no, imprime un mensaje indicando que el valor es nulo o no es un número.
  3. Declara otra variable opcional `nombreOpcional` de tipo `String` sin asignarle un valor inicial.
  4. Utiliza el operador de coalescencia nula (`??`) para imprimir el valor de `nombreOpcional` o la cadena "Usuario Anónimo" si `nombreOpcional` es `nil`.
  5. Crea una función que reciba un `String?` como parámetro y devuelva la cantidad de caracteres si el `String` no es `nil`, o 0 si es `nil`. Usa `guard let` dentro de la función.

Parte 2: Programación Orientada a Objetos (POO) en Swift

En esta sección, aplicarás los principios de la POO para modelar entidades y sus interacciones.

Ejercicio 7: Estructuras vs. Clases - Definición y Uso

- **Objetivo:** Entender las diferencias fundamentales entre estructuras y clases, y cuándo utilizar cada una.
- **Instrucciones:**
  1. Define una **estructura** llamada `Punto` con dos propiedades almacenadas: `x` e `y`, ambas de tipo `Double`.
  2. Añade un inicializador a la estructura `Punto` que reciba los valores para `x` e `y`.
  3. Añade un método a la estructura `Punto` llamado `distanciaAlOrigen()` que calcule y devuelva la distancia euclidiana desde el punto (0,0). La fórmula es x2+y2 . Puedes usar la función `sqrt()` y `pow()`.
  4. Crea una instancia de `Punto`.
  5. Define una **clase** llamada `FiguraGeometrica` con una propiedad almacenada `nombre` de tipo `String` y un método `describir()` que imprima "Soy una figura geométrica llamada [nombre]".
  6. Añade un inicializador a la clase `FiguraGeometrica` que reciba el nombre.
  7. Crea una instancia de `FiguraGeometrica`.
  8. Explica brevemente en comentarios dentro de tu código la diferencia clave entre una instancia de `Punto` (estructura) y una instancia de `FiguraGeometrica` (clase) en términos de tipos por valor vs. tipos por referencia.

Ejercicio 8: Clases, Propiedades y Métodos

- **Objetivo:** Implementar clases con propiedades (almacenadas y computadas) y métodos de instancia.
- **Instrucciones:**
  1. Define una clase llamada `Libro`.
  2. Añade las siguientes propiedades almacenadas a la clase `Libro`:
     - `titulo` (String)
     - `autor` (String)
     - `numeroPaginas` (Int)
     - `anoPublicacion` (Int)
  3. Añade una propiedad computada llamada `descripcion` (String) que devuelva una cadena formateada con el título, autor y año de

publicación del libro (ej. "El Principito por Antoine de Saint-Exupéry, publicado en 1943.").
4. Añade un inicializador para la clase `Libro` que configure todas las propiedades almacenadas.
5. Añade un método de instancia llamado `esLibroLargo()` que devuelva `true` si `numeroPaginas` es mayor a 500, y `false` en caso contrario.
6. Crea dos instancias de `Libro` con diferentes datos.
7. Imprime la `descripcion` de cada libro y si es o no un libro largo.

Ejercicio 9: Herencia y Polimorfismo

- **Objetivo:** Aplicar los conceptos de herencia para crear jerarquías de clases y polimorfismo para tratar objetos de diferentes clases de manera uniforme.
- **Instrucciones:**
  1. Define una clase base llamada `Vehiculo` con las siguientes propiedades y métodos:
     - Propiedad: `marca` (String)
     - Propiedad: `modelo` (String)
     - Inicializador que acepte `marca` y `modelo`.
     - Método: `describirVehiculo()` que imprima "Vehículo: [marca] [modelo]".
     - Método: `arrancarMotor()` que imprima "El motor del vehículo ha arrancado." (Este método será sobrescrito).
  2. Crea una clase `Coche` que herede de `Vehiculo`.
     - Añade una propiedad `numeroPuertas` (Int).
     - Sobrescribe el inicializador para incluir `numeroPuertas`. Asegúrate de llamar al inicializador de la superclase.
     - Sobrescribe el método `arrancarMotor()` para que imprima "El motor del coche [marca] [modelo] ha arrancado suavemente."
  3. Crea una clase `Motocicleta` que herede de `Vehiculo`.
     - Añade una propiedad `tipoManillar` (String, ej. "Deportivo", "Crucero").
     - Sobrescribe el inicializador para incluir `tipoManillar`. Asegúrate de llamar al inicializador de la superclase.
     - Sobrescribe el método `arrancarMotor()` para que imprima "El motor de la motocicleta [marca] [modelo] ruge al arrancar."
  4. Crea una instancia de `Coche` y una de `Motocicleta`.
  5. Crea un array llamado `misVehiculos` que contenga ambas instancias.
  6. Itera sobre el array `misVehiculos` y llama al método `arrancarMotor()` de cada elemento. Observa cómo se ejecuta la versión correcta del método para cada tipo de objeto (polimorfismo).

Ejercicio 10: Protocolos y Extensiones

- **Objetivo:** Definir y adoptar protocolos para establecer contratos de funcionalidad, y utilizar extensiones para añadir nueva funcionalidad a tipos existentes.
- **Instrucciones:**
  1. Define un protocolo llamado `Identificable` que requiera una propiedad `id` de tipo `String` (solo lectura - `{ get }`) y un método `mostrarId()` que imprima el `id`.
  2. Crea una estructura `Usuario` que adopte el protocolo `Identificable`.
     - Implementa la propiedad `id`.
     - Implementa el método `mostrarId()`.
     - Añade una propiedad `nombre` (String).
     - Añade un inicializador.
  3. Crea una clase `Producto` que también adopte el protocolo `Identificable`.
     - Implementa la propiedad `id`.
     - Implementa el método `mostrarId()`.
     - Añade una propiedad `nombreProducto` (String) y `precio` (Double).
     - Añade un inicializador.
  4. Crea una instancia de `Usuario` y una de `Producto`. Llama a `mostrarId()` en ambas.
  5. Crea una extensión para el tipo `Double` que añada un método llamado `formatoMoneda()` que devuelva el número como un `String` formateado con dos decimales y el símbolo de dólar al inicio (ej. "$19.99").
  6. Prueba tu extensión con el precio del `Producto`.

Parte 3: Ejercicio Integrador - Calculadora Básica

Objetivo: Integrar los conocimientos de Swift y POO para desarrollar una aplicación de calculadora simple que funcione en la consola.

Instrucciones:

1. **Diseño de la Calculadora:**
   - La calculadora debe ser capaz de realizar las operaciones básicas: suma, resta, multiplicación y división.
   - Debe poder manejar números decimales (`Double`).
   - Debe tener una forma de ingresar dos números y la operación deseada.
   - Debe mostrar el resultado de la operación.

- Debe manejar el caso de división por cero, mostrando un mensaje de error apropiado.

2. **Implementación (Sugerencias):**
   - Puedes crear una clase o estructura llamada `Calculadora`.
   - Esta clase/estructura podría tener métodos para cada operación: `sumar(a: Double, b: Double) -> Double`, `restar(a: Double, b: Double) -> Double`, etc.
   - Considera un método principal `calcular(numero1: Double, numero2: Double, operacion: String) -> Double?` que internamente llame al método correspondiente según la `operacion`. Devuelve un opcional para manejar errores como la división por cero o una operación inválida.
   - En el flujo principal de tu programa en `main.swift` (o el archivo principal de Replit):
     - Pide al usuario que ingrese el primer número.
     - Pide al usuario que ingrese el segundo número.
     - Pide al usuario que ingrese la operación (+, -, *, /).
     - Utiliza `readLine()` para obtener las entradas y conviértelas a los tipos adecuados (`Double` para los números). Valida que la conversión sea exitosa.
     - Crea una instancia de tu `Calculadora`.
     - Llama al método `calcular` con los datos ingresados.
     - Si el resultado es válido, imprímelo.
     - Si hubo un error (ej. división por cero), imprime un mensaje de error claro.
   - **Opcional (manejo de errores más robusto):** Puedes usar enumeraciones (`enum`) para representar las operaciones y los posibles errores, y utilizar el manejo de errores de Swift (`do-catch`, `throw`).
3. **Pruebas:**
   - Prueba cada una de las operaciones con diferentes números.
   - Prueba la división por cero.
   - Prueba ingresando operaciones no válidas.