Instituto Tecnológico de Costa Rica

Escuela de Computación

Lenguajes de Programación

Grupo 40

II Examen

Profesor (a):

Jose Rafael Castro Mora

Estudiante (s):

José Adrián Amador Ávila - 2016101574

San José, I Semestre 2023

1. Comparación de los lenguajes: Kotlin, Dart, Go (Golang), Julia

	Kotlin	**Dart**	**Go**	**Julia**
Efectos Secundarios	Es permitido pero el mismo lenguaje ofrece herramientas y conceptos para escribir código más funcional y libre de efectos secundarios. Kotlin posee variables mutables e inmutables, y aconseja usar las variables inmutables. Algunos efectos secundarios son: cambiar variables mutables, realizar operaciones de I/O o cambiar el estado de objetos.	Es permitido, pero similar a Kotlin, Dart soporta conceptos de programación funcional como funciones de orden superior y estructuras de datos inmutables. No obstante, Dart es usado para desarrollar aplicaciones móviles, web y servidores multiplataforma, donde los efectos secundarios son comúnmente usados para interactuar con las interfaces de usuario, manejar entrada y salida de datos o realizar operaciones asincrónicas.	Es permitido. Go sigue el paradigma de programación imperativo y provee características como variables mutables y soporte integrado para operaciones I/O. El modelo de concurrencia de Go, basado en go-rutinas y canales permite la comunicación entre tareas, lo cual puede involucrar efectos secundarios.	Es permitido. Julia brinda variables mutables e inmutables, además de funciones puras (programación funcional). Se puede escribir código que modifique variables, realizar operaciones I/O, etc. También se puede ver al interactuar con recursos externos como archivos, bases de datos o redes.
Transparencia referencial	No está soportado por defecto. Las funciones escritas en Kotlin pueden tener efectos secundarios y llamar a la misma función con el mismo valor de entrada puede que no genere el mismo resultado en diferentes ocasiones. Aún así, siguiendo los conceptos de	Al igual que Kotlin, en Dart se pueden escribir funciones con efectos secundarios, lo cual no permite la transparencia referencial. Pero de igual manera, siguiendo los conceptos de programación funcional se pueden escribir funciones que	De la misma manera, la transparencia referencial no está incluida para Go.	Se pueden escribir funciones siguiendo el paradigma funcional, permitiendo la transparencia referencial pero de igual manera, se pueden escribir funciones con efectos secundarios.

	Kotlin	Dart	Go	Julia
	programación funcional permitidos por Kotlin, se pueden escribir funciones sin efectos secundarios que permite la transparencia referencial.	permita la transparencia referencial.		
Sustituibilidad	Si lo aplica, ya que cuenta con los principios de la programación orientada a objetos. Se puede aprovechar la herencia y el polimorfismo para lograr la sustituibilidad. Permite la reutilización de código y la flexibilidad al trabajar con diferentes objetos.	Al igual que Kotlin, Dart también hace uso de los principios de la programación orientada a objetos, permitiendo implementar la sustituibilidad por medio de la herencia y el polimorfismo.	Go no implementa la sustituibilidad. Go soporta interfaces, lo que permite usar los tipos indistintamente pero no cuenta con herencia ni subclases como los lenguajes OOP tradicionales.	Julia sí soporta la sustituibilidad a través de su soporte de envío múltiple y herencia de tipo. Se pueden definir tipos abstractos, tipos concretos e interfaces. Los tipos abstractos se pueden usar como generalizaciones o contratos, y los tipos concretos pueden heredar de tipos abstractos u otros tipos concretos. Esta jerarquía permite la sustituibilidad, donde se puede usar un subtipo en lugar de su supertipo sin afectar la correctitud.
Paradigma	Kotlin soporta diferentes paradigmas. Principalmente se enfoca en Programación Orientada a Objetos y Funcional.	Dart es un lenguaje de programación orientado a objetos y clases.	Es un lenguaje multiparadigma: imperativo concurrente, orientado a objetos pero no de la forma convencional. Está diseñado	Julia utiliza el envío múltiple como paradigma, lo que facilita la expresión de muchos patrones de programación funcionales y orientados a

	Kotlin	Dart	Go	Julia
	Normalmente es conocido como un lenguaje pragmático. Los paradigmas soportados son: Programación Orientada a Objetos, Programación Funcional, Programación Reactiva y Programación Concurrente.		especialmente para el paradigma concurrente por sus go-rutinas y canales para la comunicación entre tareas.	objetos. También soporta: concurrencia, paralelismo y computación distribuida.
Campo de aplicación	Desarrollo de aplicaciones en Android. Desarrollo web back-end. Desarrollo web Full-stack. Ciencia de datos. Desarrollo de aplicaciones multiplataforma.	Su campo de uso abarca: aplicaciones web, servidores, aplicaciones de consola y aplicaciones móviles.	Diseñado como lenguaje de programación de sistemas, para ser usado en servidores web y arquitecturas de guardado. También en sistemas distribuidos de alto rendimiento. Por último, también puede ser usado como lenguaje de programación general para resolver problemas de procesamiento de texto o aplicaciones de scripting.	Diseñado para la aplicación en computación paralela y distribuida. Aplicado generalmente para el cómputo científico y numérico. Por ejemplo, la NASA, predicción del clima, sistemas integrados para la prevención de colisiones aéreas.

2. Implementación Puntero Inteligente en C++

```
#include <iostream>

// Clase de ejemplo
class Foo {
public:
      void bar() {
      std::cout << "Acceso a Foo" << std::endl;
      }
};

template <typename Foo>
class Puntero {
private:
      Foo* f; // Puntero crudo al objeto
      int accessCount; // Contador de acceso al objeto

public:
      Puntero(Foo* foo) : f(foo), accessCount(0) {}

      // Operador de indirección ->
      Foo* operator->() {
      accessCount++;
      return f;
      }

      // Función para obtener el número de accesos
      int getAccessCount() const {
      return accessCount;
      }

      // Destructor
      ~Puntero() {
      delete f;
      }
};

int main() {
      Puntero<Foo> p(new Foo());
```

```
    p->bar();
    p->bar();

    std::cout << "Número de accesos al objeto: " <<
p.getAccessCount() << std::endl;

    return 0;
}
```

3. Principio de Substitución de Liskov en Scala.

En Scala, el principio de substitución de Liskov es aplicado a través del sistema de tipos y el concepto de subtipos. El sistema de subtipos permite definir jerarquías de clases y establecer relaciones de subtipos. Scala provee lo siguiente para el principio:

1. Herencia de clases.
2. Jerarquía de tipos: forma la base para los subtipos y el principio de substitución. En la parte superior de la jerarquía el tipo 'Any', es la superclase de todos los demás tipos en Scala. Tiene dos subclases directas: 'AnyVal' (para tipos de valores) y 'AnyRef' (para tipos de referencia).
3. Covarianza y contravarianza: permite especificar cómo varían los tipos con respecto a los subtipos. Covarianza se denota con '+' y contravarianza con '-'.
4. Límites de tipo: restringe aún más los tipos que se pueden utilizar en un contexto dado. Por ejemplo, 'A <: B' es un subtipo de B (Covarianza) o 'A >: B' el tipo A es un supertipo de B (Contravarianza).

A continuación, se muestra un ejemplo de código en Scala usando estos conceptos. La clase abstracta 'Animal' es la clase 'padre' de las cuales 'Dog' y 'Cat' extienden (hijas) (concepto de herencia). Dentro de las subclases, se sobreescribe el método 'sound'. En la clase 'AnimalCollection' se especifica la covarianza '+A' para permitir subtipificación. Toma un parámetro 'A' es cuál debe ser subtipo de 'Animal', es decir, '+A <: Animal'. El método 'add' utiliza los límites de tipo 'B >: A <: Animal' para asegurar que el animal agregado es supertipo de 'A' y subtipo de 'Animal'. Lo demás del código es fácil de seguir.

```scala
abstract class Animal {
  def sound: String
}

class Dog extends Animal {
  override def sound: String = "Woof"
}

class Cat extends Animal {
  override def sound: String = "Meow"
}

class AnimalCollection[+A <: Animal](animals: List[A]) {
  def add[B >: A <: Animal](animal: B): AnimalCollection[B] =
      new AnimalCollection[B](animal :: animals)

  def makeSound(): Unit = animals.foreach(animal => println(animal.sound))
}

object LiskovSubsitutionExample {
  def main(args: Array[String]): Unit = {
      val dog: Dog = new Dog()
      val cat: Cat = new Cat()

      val animalCollection: AnimalCollection[Animal] =
                              new AnimalCollection[Animal](List.empty)
      val dogCollection: AnimalCollection[Dog] =
                              new AnimalCollection[Dog](List.empty)

      val mixedCollection: AnimalCollection[Animal] =
                              animalCollection.add(dog)
      val mixedCollection2: AnimalCollection[Animal] =
                              animalCollection.add(cat)

      val dogCollection2: AnimalCollection[Dog] =
                              dogCollection.add(dog)

      mixedCollection.makeSound()
      mixedCollection2.makeSound()
      dogCollection2.makeSound()
  }
}
```

4. Julia Just in Time
    1. Julia logra esto de la siguiente manera:
        a. Durante la compilación, Julia realiza inferencia de tipos para determinar los tipos de todas las variables y expresiones en el código.
        b. La estabilidad de tipos está presente en las funciones, el compilador puede saber el tipo de las variables en cada paso. Multiple-dispatch (envíos múltiples) asegura que una función pueda ser de tipo estable: solo significa diferentes cosas para diferentes entradas. Pero como el compilador sabe (infiere) los tipos de las variables antes de llamar a la función, entonces sabe cuál de todas las implementaciones de dicha función usar.
        c. En cuanto a la estructura de la biblioteca de clases de Julia, Julia se basa en un diseño de tipos compuesto por una combinación de tipos abstractos y tipos concretos. Los tipos abstractos son utilizados como interfaces, mientras que los tipos concretos son implementaciones específicas. Los tipos en Julia pueden ser subtipos de otros tipos, lo que permite la herencia y la composición.
    2. Segunda pregunta:
        a. Una clase en Julia que no sea virtual puede tener subclases. Esto se debe a que Julia no utiliza el enfoque de herencia de clases tradicional que se encuentra en otros lenguajes orientados a objetos como Java o C + +.
        b. Los tipos abstractos se utilizan como interfaces para definir contratos. Las subclases pueden implementar estos contratos a través de la herencia y la implementación de los métodos necesarios. Esta decisión de diseño está relacionada con el principio de sustitución de Liskov. Julia al usar los tipos abstractos y definir contactos a través de interfaces, garantiza que las subclases cumplan con los mismos comportamientos y contratos que de los tipos abstractos superiores.