**Polimorfismo en Dart**

El polimorfismo es un concepto de la programación orientada a objetos, permite que una función o un método se comporte de manera diferente según el objeto que lo llame.

En Dart, este concepto se implementa de diferentes maneras, incluyendo el uso de clases abstractas, mixins y la anotación @override.

## **¿Qué es el Polimorfismo?**

El término "polimorfismo" proviene del griego y significa "POLI - Muchos/Muchas" “MORFISMO -Formas/Composición de algo”. En programación, el polimorfismo se refiere a la capacidad de diferentes clases para ser tratadas como instancias de la misma clase a través de una interfaz común. Esto permite que una función use objetos de diferentes clases de manera intercambiable.

### **Conceptos Clave**

1. **Clase Base y Subclases**: Una clase base (o clase padre) define métodos que pueden ser sobrescritos por sus subclases (o clases hijas).
2. **Sobrescritura de Métodos**: Las subclases pueden sobrescribir métodos definidos en la clase base para proporcionar una implementación específica.
3. **Interfaces y Clases Abstractas**: Las interfaces y las clases abstractas se utilizan para definir contratos que las clases concretas deben seguir.

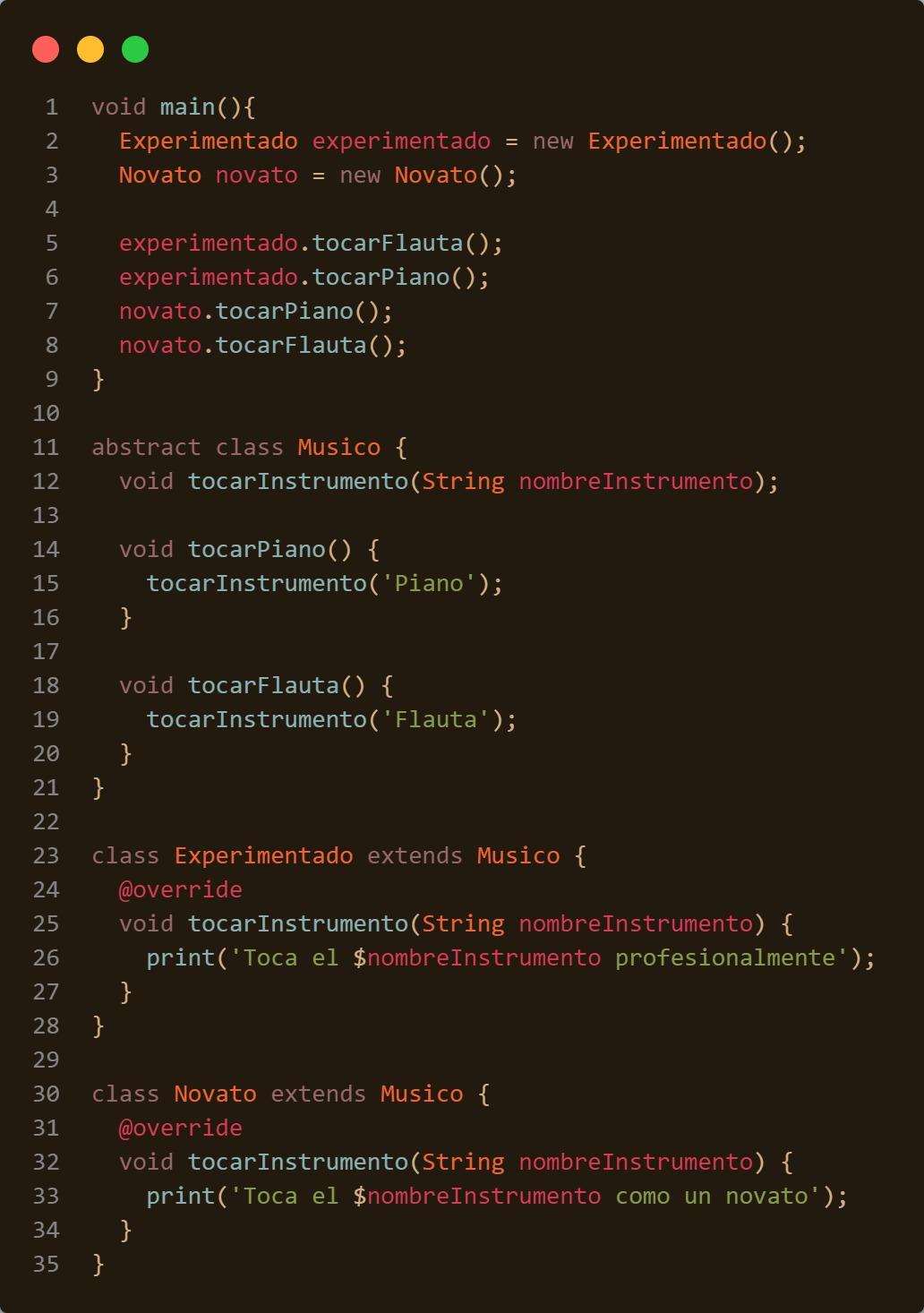
## **¿Cuándo y Cómo Utilizar el Polimorfismo?**

El polimorfismo es útil cuando queremos que diferentes clases compartan la misma interfaz y podamos tratarlas de manera uniforme sin preocuparnos por su implementación específica. Se utiliza comúnmente en:

* **Diseño de API**: Permite diseñar interfaces que pueden ser implementadas por múltiples clases.
* **Simplificación de Código**: Facilita la escritura de código genérico que funciona con cualquier clase que implemente la interfaz esperada.
* **Extensibilidad**: Permite agregar nuevas clases que implementan la interfaz sin cambiar el código existente.

## **Clases Abstractas**

Las clases abstractas en Dart se utilizan para definir una interfaz común para un grupo de subclases. No se pueden instanciar directamente, lo que significa que no puedes crear objetos de una clase abstracta. Las clases abstractas pueden contener métodos que no tienen una implementación (métodos abstractos) y que deben ser implementados por las subclases.



**Clase Abstracta: Musico**

* **Definición**: Musico es una clase abstracta, lo que significa que no puede ser instanciada directamente. Sirve como una plantilla para otras clases.
* **Métodos Abstractos**: tocarInstrumento es un método abstracto que no tiene implementación en la clase Musico. Esto obliga a las clases derivadas (Experimentado y Novato) a proporcionar su propia implementación de este método.
* **Métodos Concretos**: tocarPiano y tocarFlauta son métodos concretos que sí tienen una implementación en Musico. Estos métodos llaman al método abstracto tocarInstrumento con el nombre del instrumento correspondiente.

**Clase Subclase: Experimentado**

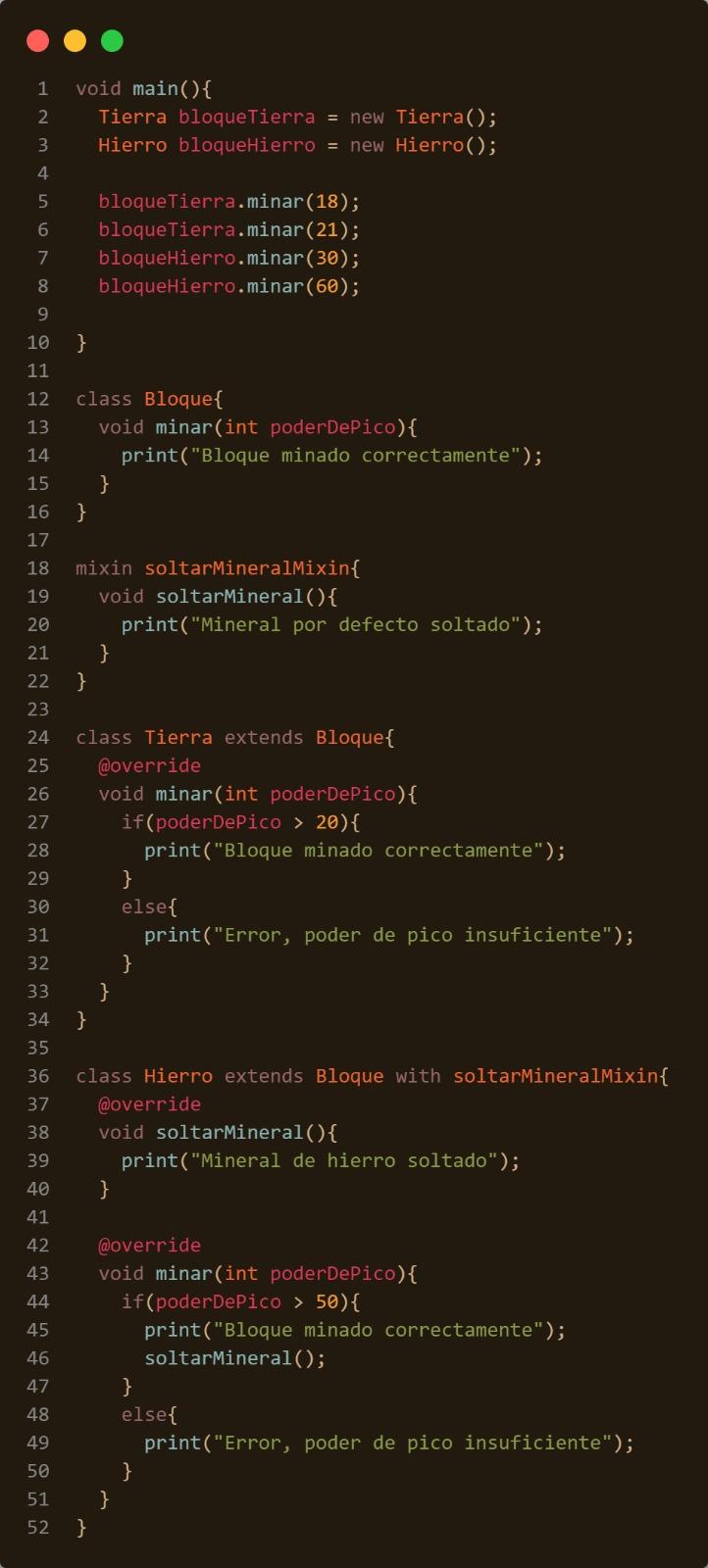
* **Sobrescritura del Método Abstracto**: Experimentado extiende(Hereda de) la clase Musico y proporciona una implementación del método tocarInstrumento usando la anotación @override. En este caso, imprime "Toca el [nombre del instrumento] profesionalmente".

**Clase Derivada: Novato**

* **Sobrescritura del Método Abstracto**: Novato también extiende Musico y proporciona su propia implementación del método tocarInstrumento. Aquí, imprime "Toca el [nombre del instrumento] como un novato".

## **Mixins**

Los mixins en Dart son una forma de reutilizar código en múltiples clases. Los mixins te permiten definir métodos que pueden ser incluidos en cualquier clase utilizando la palabra clave with. Un mixin se define como una clase normal, pero generalmente no tiene un constructor y está pensado para ser utilizado junto con otras clases.



**Clase Base: Bloque**

* La clase Bloque tiene un método minar que imprime "Bloque minado correctamente".
* Esta es una implementación básica que las clases derivadas pueden sobrescribir.

**Mixin: soltarMineralMixin**

* Un mixin es una clase que proporciona métodos para ser usados por otras clases sin ser una clase base de esas clases.
* En este ejemplo, el mixin soltarMineralMixin define un método soltarMineral que imprime "Mineral por defecto soltado".
* Los mixins se utilizan para compartir funcionalidades entre clases de una manera que no se puede lograr solo con herencia simple.

**Clase Subclase: Tierra**

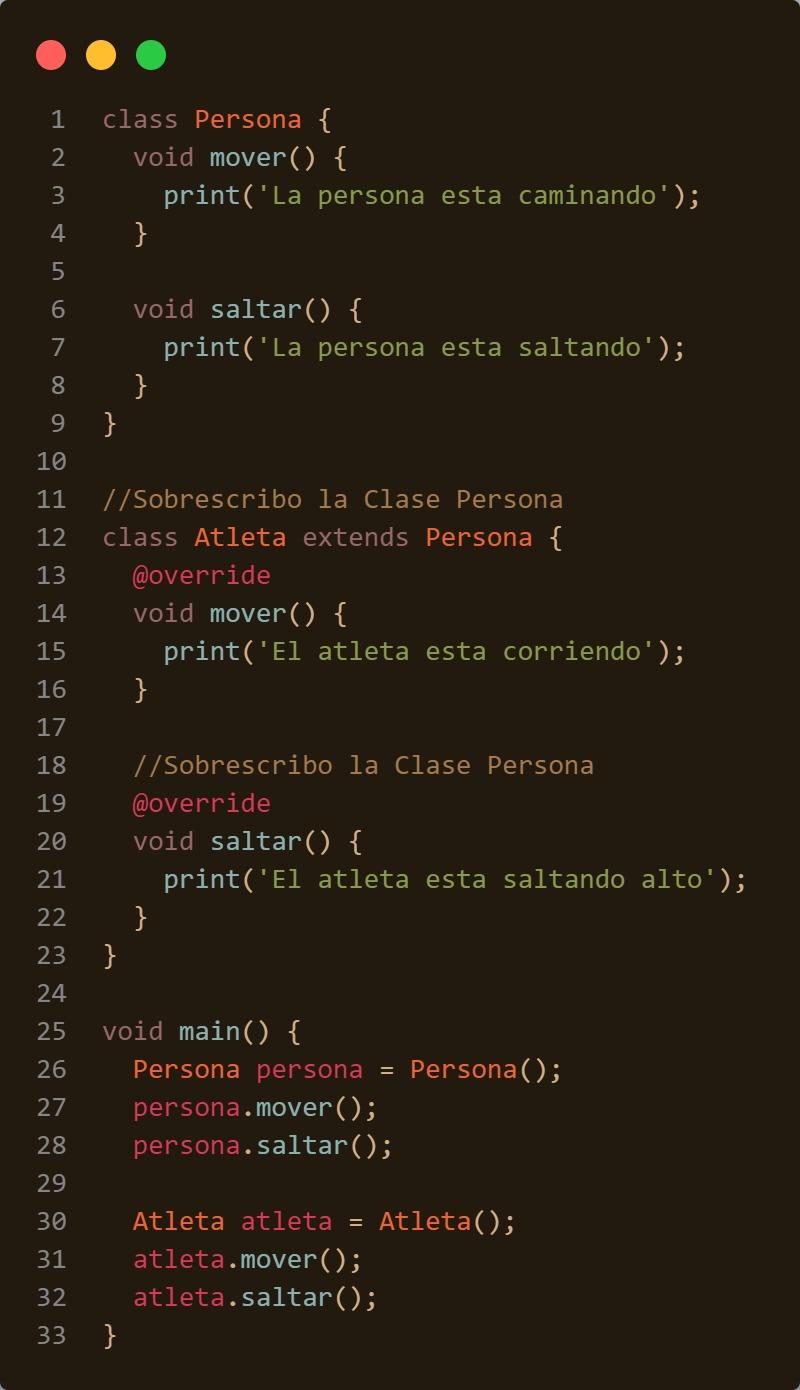
* La clase Tierra extiende Bloque y sobrescribe el método minar.
* En Tierra, el método minar imprime "Bloque minado correctamente" si el poderDePico es mayor que 20; de lo contrario, imprime "Error, poder de pico insuficiente".

**Clase Subclase: Hierro**

* La clase Hierro extiende Bloque y utiliza el mixin soltarMineralMixin.
* Utilizando el mixin, Hierro adquiere el método soltarMineral.
* Además, sobrescribe el método soltarMineral para proporcionar una implementación específica: "Mineral de hierro soltado".
* También sobrescribe el método minar. Si el poderDePico es mayor que 50, imprime "Bloque minado correctamente" y llama al método soltarMineral; de lo contrario, imprime "Error, poder de pico insuficiente".

## **@override**

La anotación @override en Dart se utiliza para indicar que un método en una subclase está sobrescribiendo un método de la clase base.



## **Clase Base: Persona**

## La clase Persona tiene dos métodos: mover y saltar.

## El método mover imprime "La persona está caminando".

## El método saltar imprime "La persona está saltando".

## **Clase Subclase: Atleta**

## La clase Atleta extiende (hereda de) la clase Persona.

## Utilizamos la anotación @override para sobrescribir los métodos mover y saltar de la clase base Persona.

## En Atleta, el método mover imprime "El atleta está corriendo".

## En Atleta, el método saltar imprime "El atleta está saltando alto".

## La anotación @override se utiliza para indicar que un método en la clase Subclase (Atleta) está sobrescribiendo un método de la clase base (Persona).

## 

## **Conclusión**

El polimorfismo en Dart permite escribir código más flexible y reutilizable. Las clases abstractas, los mixins y la anotación @override son herramientas poderosas para implementar el polimorfismo. Utilizando estos conceptos, podemos diseñar sistemas que sean más fáciles de mantener y extender.